HCL Informix 15.0.0

SQL programming

Contents

Chapter 1. SQL programming.....3
Guide to SQL: Syntax.....3
Overview of SQL syntax 3
SQL statements.....18
SPL statements.....911
Data types and expressions.....963
Other syntax segments.....1209
Guide to SQL: Syntax.....
Appendixes.....1295
Guide to SQL: Reference.....1315
System catalog tables.....1316
Data types.....1389
Environment variables.....1440
Appendixes.....1516
Guide to SQL: Tutorial.....1520
Database concepts.....1521
Compose SELECT statements.....1532
Select data from complex types.....1580
Functions in SELECT statements.....1591
Compose advanced SELECT statements.....1619
Modify data.....1658
Access and modify data in an external database.....1696
SQL programming.....1700
Modify data through SQL programs.....1723
Programming for a multiuser environment.....1733
Create and use SPL routines.....1753
Create and use triggers.....1821
Index.....1835

Chapter 1. SQL programming

You can use the HCL Informix® implementation of the SQL language to develop applications for Informix® database servers.

Guide to SQL: Syntax

The *HCL® Informix® Guide to SQL: Syntax* describes the syntax of the statements, data types, expressions, operators, and built-in functions of the Informix® dialect of the SQL language.

This information is intended for the following users:

- Database users
- Database administrators
- Database security administrators
- Database application programmers.

This information assumes that you have the following background:

- A working knowledge of your computer, your operating system, and the utilities that your operating system provides
- Some experience working with relational databases or exposure to database concepts
- Some experience with computer programming.

Overview of SQL syntax

These topics provide an overview about how to use the SQL statements, SPL statements, and syntax segments.

The topics in this chapter are organized into the following sections.

Section	Scope
How to Enter SQL Statements on page 3	How to use syntax diagrams and descriptions to enter SQL statements correctly
How to Enter SQL Comments on page 6	How to enter comments in SQL statements
Categories of SQL Statements on page 8	The SQL statements, listed by functional category
ANSI/ISO Compliance and Extensions on page 14	The SQL statements, listed by degree of ANSI/ISO compliance

How to Enter SQL Statements

SQL is a free-form language, like C or PASCAL, that generally ignores white-space characters like TAB, LINEFEED, and extra blank spaces between statements or statement elements. At least one blank character or other delimiter, however, must separate keywords and identifiers from other syntax tokens.

SQL is lettercase insensitive, except within quoted strings; see also Identifier on page 1231. In an ANSI-compliant database, if you do not delimit the *owner* of an object by double (") quotation marks, and the **ANSIOWNER** environment variable was not set to 1 when the database server was initialized, the database server stores the *owner* name in uppercase letters.

Statement descriptions are provided in this publication to help you to enter SQL statements successfully. A statement description includes this information:

- A brief introduction that explains what the statement does
- A syntax diagram that shows how to enter the statement correctly
- A syntax table that explains each input parameter in the syntax diagram
- Rules of usage, typically with examples that illustrate these rules

For some statements, this information is provided for individual clauses.

Most statement descriptions conclude with references to related information in this publication and in other publications.

SQL statements on page 18 provides descriptions of each SQL statement, arranged in alphabetical order. SPL statements on page 911 describes each of the SPL statements, using the same format.

The major aids for entering SQL statements include:

- The combination of the syntax diagram and syntax table
- The examples of syntax that appear in the rules of usage
- The references to related information

Using Syntax Diagrams and Syntax Tables

Before you try to use the syntax diagrams in this chapter, it is helpful to read the syntax diagram section of the Introduction. This section is the key to understanding the syntax diagrams and explains the elements that can appear in a syntax diagram and the paths that connect the elements to each other. This section also includes an example that illustrates the elements of typical syntax diagrams. The narrative that follows the example diagram shows how to read the diagram in order to enter the statement successfully.

Notes® to the diagram can reference other syntax segments or can specify various restrictions. If you are using an application programming interface, such as ESQL/C, only the SQL syntax rules that both your client application and the database server support are valid.

When a syntax diagram includes input specifications that are not keywords, syntax segments, nor punctuation symbols, such as identifiers, expressions, filenames, or host variables, the syntax diagram is followed by a table that describes how to enter the term without generating errors. Each syntax table includes four columns:

- The **Element** column lists each variable term in the syntax diagram.
- The **Description** column briefly describes the term and identifies the default value, if the term has one.

- The **Restrictions** column summarizes the restrictions on the term, such as acceptable ranges of values. (For some diagrams, restrictions that cannot be tersely summarized appear in the **Usage** notes, rather than in this column.)
- The **Syntax** column points to the SQL segment that gives the detailed syntax for the term. For a few terms, such as the names of host variables, pathnames, or literal characters, no page reference is provided.

The diagrams generally provide an intuitive notation for what is valid in a given SQL statement, but for some statements, dependencies or restrictions among syntax elements are identified only in the text of the Usage section.

Using Examples

To understand the main syntax diagram and subdiagrams for a statement, study the examples of syntax that appear in the rules of usage for each statement. These examples have two purposes:

- To show how to accomplish specific tasks with the statement or its clauses
- To show how to use syntax of the statement or its clauses in a concrete way

Tip: An efficient way to understand a syntax diagram is to find an example of the syntax and compare it with the keywords and parameters in the syntax diagram. By mapping the concrete elements of the example to the abstract elements of the syntax diagram, you can understand the syntax diagram and use it more effectively.

For an explanation of the conventions used in the examples in this publication, see the syntax diagram section of the Introduction.

These code examples are program fragments to illustrate valid syntax, rather than complete SQL programs. In some code examples, ellipsis (. . .) symbols indicate that additional code has been omitted. To save space, however, ellipses are not shown at the beginning or end of the program fragments.

Using Related Information

For help in understanding concepts and terms in the SQL statement description, check the “Related Information” section at the end of each statement.

This section points to related information in this publication and other publications to help you understand the statement in question. The section provides some or all of the following information:

- The names of related statements that might contain a fuller discussion of topics in this statement
- The titles of other publications that provide extended discussions of topics in this statement

Tip: If you do not have extensive knowledge and experience with SQL, the *HCL® Informix® Guide to SQL: Tutorial* gives you the basic SQL knowledge that you need to understand and use the statement descriptions in this document.

How to Enter SQL Comments

You can add comments to clarify the purpose or effect of particular SQL statements. You can also use comment symbols during program development to disable individual statements without deleting them from your source code.

Your comments can help you or others to understand the role of the statement within a program, SPL routine, or command file. The code examples in this document sometimes include comments that clarify the role of an SQL statement within the code, but your own SQL programs will be easier to read and to maintain if you document them with frequent comments.

The following table shows the SQL comment indicators that you can enter in your code. Here a *Y* in a column signifies that you can use the symbol with the product or with the type of database identified in the column heading. An *N* in a column signifies that you cannot use the symbol with the indicated product or with a database of the indicated ANSI-compliance status.

Comment Symbol	ESQL/C	SPL Routine		ANSI-Compliant Databases	Databases Not ANSI Compliant	Description
double hyphen (--)	Y	Y	Y	Y	Y	The double hyphen precedes a comment within a single line. To comment more than one line, put double hyphen symbols at the beginning of each comment line.
braces ({ . . . })	N	Y	Y	Y	Y	Braces enclose the comment. The { precedes the comment, and the } follows it. Braces can delimit single- or multiple-line comments, but comments cannot be nested.
slash and asterisk / * . . . */	Y	Y	Y	Y	Y	C-language style slash and asterisk (/* */) paired delimiters enclose the comment. The /* precedes the comment, and the */ follows it. These can delimit single-line or multiple-line

Comment Symbol	ESQL/C	SPL Routine	ANSI-Compliant Databases	Databases Not ANSI Compliant	Description
					comments, but comments cannot be nested.

Characters within the comment are ignored by the database server.

The section Optimizer Directives on page 1246 describes a context where information within comments can influence query plans of Informix®.

If the product that you use supports all of these comment symbols, your choice of a comment symbol depends on requirements for ANSI/ISO compliance:

- Double hyphen (`--`) complies with the ANSI/ISO standard for SQL.
- Braces ({ }) are the Informix® extension to the ANSI/ISO standard.
- C-style slash-and-asterisk (/* . . . */) comply with the SQL-99 standard.

If ANSI/ISO compliance is not an issue, your choice of comment symbols is a matter of personal preference.

In , you can use any of these comment symbols when you enter SQL statements with the SQL editor and when you create SQL command files with the SQL editor or with a system editor.

An SQL command file is an operating-system file that contains one or more SQL statements. Command files are also known as command scripts. For more information about command files, see the discussion of command scripts in the *HCL® Informix® Guide to SQL: Tutorial*. For information on how to create and modify command files with the SQL editor or a system editor in , see the *HCL® Informix® DB-Access User's Guide*.

You can use any of these comment symbols in any line of an SPL routine. See the discussion of how to comment and document an SPL routine in the *HCL® Informix® Guide to SQL: Tutorial*.

In , the double hyphen (`--`) can begin a comment that extends to the end of the same line. For information on language-specific comment symbols in programs, see the *HCL® Informix® Enterprise Replication Guide*.

Examples of SQL Comments

These examples illustrate different ways to use the SQL comment indicators.

The following examples use each style of comment indicator, including the double hyphen (`--`), braces (`{ }`), and C-style (`/* . . . */`) comment delimiters to include a comment after an SQL statement. The comment appears on the same line as the statement.

```
SELECT * FROM customer; -- Selects all columns and rows

SELECT * FROM customer; {Selects all columns and rows}

SELECT * FROM customer; /*Selects all columns and rows*/
```

The next three examples use the same SQL statement and the same comments as in the preceding examples, but place the comment on a separate line:

```
SELECT * FROM customer;
   -- Selects all columns and rows

SELECT * FROM customer;
   {Selects all columns and rows}

SELECT * FROM customer;
   /*Selects all columns and rows*/
```

In the following examples, the user enters the same SQL statement as in the preceding example but now a multiple-line comment (or for the double-hyphen indicator, two comments) follows each statement:

```
SELECT * FROM customer;
   -- Selects all columns and rows
   -- from the customer table

SELECT * FROM customer;
   {Selects all columns and rows
    from the customer table}

SELECT * FROM customer;
   /*Selects all columns and rows
    from the customer table*/
```

Comments in any of these styles can also appear within an SQL statement:

```
SELECT *           -- Selects all columns and rows
   FROM customer;  -- from the customer table

SELECT *           {Selects all columns and rows}
   FROM customer;  {from the customer table}

SELECT *           /*Selects all columns and rows*/
   FROM customer;  /*from the customer table*/
```

If you use braces or C-style comments that are delimited by paired opening and closing indicators, the closing comment indicator must be in the same style as the opening comment indicator.

Non-ASCII Characters in SQL Comments

You can enter non-ASCII characters (including multibyte characters) in SQL comments if the database locale supports the non-ASCII characters. For further information on the aspects of SQL comments, see the *HCL® Informix® GLS User's Guide*.

Categories of SQL Statements

SQL statements are traditionally divided into the following logical categories:

Data definition statements

These data definition language (DDL) statements can declare, rename, modify, or destroy objects in the local database.

Data manipulation statements

These data manipulation language (DML) statements can retrieve, insert, delete, or modify data values.

Cursor manipulation statements

These statements can declare, open, and close cursors, which are data structures for operations on multiple rows of data.

Dynamic management statements

These statements support memory management and allow users to specify at runtime the details of DML operations.

Data access statements

These statements specify discretionary access privileges and support concurrent access to the database by multiple users.

Data integrity statements

These implement transaction logging and support the referential integrity of the database.

Optimization statements

These can be used to improve the performance of operations on the database.

Routine definition statements

These can declare, define, modify, execute, or destroy user-defined routines that the database stores.

Client/server connection statements

These can open or close a connection between a database and a client application.

Auxiliary statements

These can provide information about the database. (This is also a residual category for statements that are not closely related to the other statement categories.)

Data Definition Language Statements

The data definition language (DDL) statements of SQL create, modify, rename, or destroy database objects, and make corresponding changes to rows in the system catalog tables of the database.

- ALTER ACCESS_METHOD
- ALTER FRAGMENT
- ALTER FUNCTION
- ALTER INDEX
- ALTER PROCEDURE
- ALTER ROUTINE
- ALTER SECURITY LABEL COMPONENT
- ALTER SEQUENCE
- ALTER TABLE
- ALTER TRUSTED CONTEXT

- ALTER USER
- CLOSE DATABASE
- CREATE ACCESS_METHOD
- CREATE AGGREGATE
- CREATE CAST
- CREATE DATABASE
- CREATE DEFAULT USER
- CREATE DISTINCT TYPE
- CREATE EXTERNAL TABLE
- CREATE FUNCTION
- CREATE FUNCTION FROM
- CREATE INDEX
- CREATE OPAQUE TYPE
- CREATE OPCLASS
- CREATE PROCEDURE
- CREATE PROCEDURE FROM
- CREATE ROLE
- CREATE ROUTINE FROM
- CREATE ROW TYPE
- CREATE SCHEMA
- CREATE SECURITY LABEL
- CREATE SECURITY LABEL COMPONENT
- CREATE SECURITY POLICY
- CREATE SEQUENCE
- CREATE SYNONYM
- CREATE TABLE
- CREATE TEMP TABLE
- CREATE TRIGGER
- CREATE TRUSTED CONTEXT
- CREATE USER
- CREATE VIEW
- CREATE XADATASOURCE
- CREATE XADATASOURCE TYPE
- DROP ACCESS_METHOD
- DROP AGGREGATE
- DROP CAST
- DROP DATABASE
- DROP FUNCTION
- DROP INDEX
- DROP OPCLASS
- DROP PROCEDURE
- DROP ROLE

- DROP ROUTINE
- DROP ROW TYPE
- DROP SECURITY
- DROP SEQUENCE
- DROP SYNONYM
- DROP TABLE
- DROP TRIGGER
- DROP TRUSTED CONTEXT
- DROP TYPE
- DROP USER
- DROP VIEW
- DROP XADATASOURCE
- DROP XADATASOURCE TYPE
- RENAME COLUMN
- RENAME DATABASE
- RENAME INDEX
- RENAME SECURITY
- RENAME SEQUENCE
- RENAME TABLE
- RENAME TRUSTED CONTEXT
- RENAME USER
- TRUNCATE
- UPDATE STATISTICS

Data Manipulation Language Statements

- DELETE
- INSERT
- LOAD
- MERGE
- SELECT
- UNLOAD
- UPDATE

Note: DELETE, INSERT, MERGE, SELECT, and UPDATE are DML statements in the ANSI/ISO standard for SQL, where MERGE can emulate INSERT and DELETE or UPDATE. Although LOAD and UNLOAD resemble DML in their functionality, these macros are out-of-scope for most references in this document to "DML statements."

Data Integrity Statements

- BEGIN WORK
- COMMIT WORK

- SAVEPOINT
- RELEASE SAVEPOINT
- ROLLBACK WORK
- SET Database Object Mode
- SET LOG
- SET Transaction Mode
- START VIOLATIONS TABLE
- STOP VIOLATIONS TABLE

Cursor Manipulation Statements

- CLOSE
- DECLARE
- FETCH
- FLUSH
- FREE
- OPEN
- PUT
- SET AUTOFREE

Dynamic Management Statements

- ALLOCATE COLLECTION
- ALLOCATE DESCRIPTOR
- ALLOCATE ROW
- DEALLOCATE COLLECTION
- DEALLOCATE DESCRIPTOR
- DEALLOCATE ROW
- DESCRIBE
- DESCRIBE INPUT
- EXECUTE
- EXECUTE IMMEDIATE
- FREE
- GET DESCRIPTOR
- INFO
- PREPARE
- SET DEFERRED_PREPARE
- SET DESCRIPTOR

Data Access Statements

- GRANT
- GRANT FRAGMENT

- LOCK TABLE
- REVOKE
- REVOKE FRAGMENT
- SET ISOLATION
- SET LOCK MODE
- SET ROLE
- SET SESSION AUTHORIZATION
- SET TRANSACTION
- SET Transaction Mode
- UNLOCK TABLE

Optimization Statements

- SAVE EXTERNAL DIRECTIVES
- SET ENVIRONMENT
- SET EXPLAIN
- SET OPTIMIZATION
- SET PDQPRIORITY
- SET STATEMENT CACHE

Routine Definition Statements

- ALTER FUNCTION
- ALTER PROCEDURE
- ALTER ROUTINE
- CREATE FUNCTION
- CREATE FUNCTION FROM
- CREATE PROCEDURE
- CREATE PROCEDURE FROM
- CREATE ROUTINE FROM
- DROP FUNCTION
- DROP PROCEDURE
- DROP ROUTINE
- EXECUTE FUNCTION
- EXECUTE PROCEDURE
- SET DEBUG FILE TO

Auxiliary Statements

- GET DIAGNOSTICS
- INFO
- OUTPUT
- SET COLLATION

- SET DATASKIP
- SET ENCRYPTION PASSWORD
- SET USER PASSWORD
- WHENEVER

Client/Server Connection Statements

- CONNECT
- DATABASE
- DISCONNECT
- SET CONNECTION

ANSI/ISO Compliance and Extensions

Lists that follow show statements that match the ANSI SQL-92 standard at the entry level, statements that are ANSI compliant but include Informix® extensions, and statements that are Informix® extensions to the ANSI/ISO standard.

ANSI/ISO-Compliant Statements

- CLOSE
- COMMIT WORK
- RELEASE SAVEPOINT
- SET CONSTRAINTS (See SET Transaction Mode statement on page 841)
- SET SESSION AUTHORIZATION
- SET TRANSACTION

ANSI/ISO-Compliant Statements with Informix® Extensions

- ALLOCATE DESCRIPTOR
- ALTER TABLE
- CONNECT
- CREATE FUNCTION
- CREATE PROCEDURE
- CREATE TRIGGER
- CREATE SCHEMA
- CREATE TABLE
- CREATE TEMP TABLE
- CREATE VIEW
- DEALLOCATE DESCRIPTOR
- DECLARE
- DELETE
- DESCRIBE
- DESCRIBE INPUT

- DISCONNECT
- EXECUTE
- EXECUTE IMMEDIATE
- FETCH
- GET DESCRIPTOR
- GET DIAGNOSTICS
- GRANT
- INSERT
- MERGE
- OPEN
- PREPARE
- REVOKE
- ROLLBACK WORK
- SAVEPOINT
- SELECT
- SET CONNECTION
- SET DESCRIPTOR
- UPDATE STATISTICS
- WHENEVER

Statements that are Extensions to the ANSI/ISO Standard

- ALLOCATE COLLECTION
- ALLOCATE ROW
- ALTER ACCESS_METHOD
- ALTER FRAGMENT
- ALTER FUNCTION
- ALTER INDEX
- ALTER PROCEDURE
- ALTER ROUTINE
- ALTER SECURITY LABEL COMPONENT
- ALTER SEQUENCE
- ALTER TRUSTED CONTEXT
- ALTER USER
- BEGIN WORK
- CLOSE DATABASE
- CREATE ACCESS_METHOD
- CREATE AGGREGATE
- CREATE CAST
- CREATE DATABASE
- CREATE DEFAULT USER
- CREATE DISTINCT TYPE

- CREATE EXTERNAL TABLE
- CREATE FUNCTION
- CREATE FUNCTION FROM
- CREATE INDEX
- CREATE OPAQUE TYPE
- CREATE OPCLASS
- CREATE PROCEDURE
- CREATE PROCEDURE FROM
- CREATE ROLE
- CREATE ROUTINE FROM
- CREATE ROW TYPE
- CREATE SECURITY LABEL
- CREATE SECURITY LABEL COMPONENT
- CREATE SECURITY POLICY
- CREATE SEQUENCE
- CREATE SYNONYM
- CREATE TRIGGER
- CREATE TRUSTED CONTEXT
- CREATE USER
- CREATE XADATASOURCE
- CREATE XADATASOURCE TYPE
- DATABASE
- DEALLOCATE COLLECTION
- DEALLOCATE ROW
- DROP ACCESS_METHOD
- DROP AGGREGATE
- DROP CAST
- DROP DATABASE
- DROP FUNCTION
- DROP INDEX
- DROP OPCLASS
- DROP PROCEDURE
- DROP ROLE
- DROP ROUTINE
- DROP ROW TYPE
- DROP SECURITY LABEL
- DROP SECURITY LABEL COMPONENT
- DROP SECURITY POLICY
- DROP SEQUENCE
- DROP SYNONYM
- DROP TABLE
- DROP TRIGGER

- DROP TRUSTED CONTEXT
- DROP TYPE
- DROP USER
- DROP VIEW
- DROP XADATASOURCE
- DROP XADATASOURCE TYPE
- EXECUTE FUNCTION
- EXECUTE PROCEDURE
- FLUSH
- FREE
- GRANT FRAGMENT
- LOAD
- LOCK TABLE
- OUTPUT
- PUT
- RELEASE SAVEPOINT
- RENAME COLUMN
- RENAME DATABASE
- RENAME INDEX
- RENAME SECURITY LABEL
- RENAME SECURITY LABEL COMPONENT
- RENAME SECURITY POLICY
- RENAME SEQUENCE
- RENAME TABLE
- RENAME TRUSTED CONTEXT
- RENAME USER
- REVOKE FRAGMENT
- SAVE EXTERNAL DIRECTIVES
- SET AUTOFREE
- SET COLLATION
- SET CONSTRAINTS (See SET Database Object Mode statement on page 784.)
- SET Database Object Mode
- SET DATASKIP
- SET DEBUG FILE TO
- SET DEFERRED_PREPARE
- SET ENCRYPTION PASSWORD
- SET ENVIRONMENT
- SET EXPLAIN
- SET INDEXES (See SET Database Object Mode statement on page 784.)
- SET ISOLATION
- SET LOCK MODE
- SET LOG

- SET OPTIMIZATION
- SET PDQPRIORITY
- SET ROLE
- SET STATEMENT CACHE
- SET TRIGGERS (See SET Database Object Mode statement on page 784.)
- SET USER PASSWORD
- START VIOLATIONS TABLE
- STOP VIOLATIONS TABLE
- TRUNCATE
- UNLOAD
- UNLOCK TABLE
- UPDATE STATISTICS

SQL statements

This chapter describes the syntax and semantics of SQL statements that are recognized by Informix®.

The SQL statement names that appear as the title to each statement description in this chapter are listed in alphabetical order.

For some statements, important details of the semantics appear in other volumes of this documentation set, as indicated by cross-references.

For many statements, the syntax diagram, or the table of terms immediately following the diagram, or both, includes references to syntax segments in Data types and expressions on page 963 or in Other syntax segments on page 1209.

When the name of an SQL statement includes lowercase characters, such as SET Database Object Mode, it means that the first mixed-lettercase string in the statement name is not an SQL keyword, but that two or more different SQL keywords can follow the preceding uppercase keyword.

For an explanation of the structure of statement descriptions, see Overview of SQL syntax on page 3.

ALLOCATE COLLECTION statement

Use the ALLOCATE COLLECTION statement to allocate memory for a variable of a collection data type (such as LIST, MULTISET, or SET) or for an untyped collection variable.

Syntax

```
ALLOCATE COLLECTION variable
```

Element	Description	Restrictions	Syntax
variable	Name of the typed or untyped collection variable to allocate	Must be an unallocated collection-type host variable	Language-specific rules for names

Usage

This statement is an extension to the ANSI/ISO standard for SQL. Use this statement with ESQL/C.

The ALLOCATE COLLECTION statement allocates memory for an ESQL/C variable that can store the value of a collection data type.

To create a collection variable for programs:

1. Declare the collection variable as a client collection variable in the program.

   The collection variable can be a typed or untyped collection variable.
2. Allocate memory for the collection variable with the ALLOCATE COLLECTION statement.

The ALLOCATE COLLECTION statement sets **SQLCODE** (that is, **sqlca.sqlcode**) to zero (0) if the memory allocation was successful, or to a negative error code if the allocation failed.

When you no longer need the collection variable, you must explicitly release the memory that it occupies with the DEALLOCATE COLLECTION statement. After the DEALLOCATE COLLECTION statement executes successfully, you can reuse the collection variable.

Tip: The ALLOCATE COLLECTION statement allocates memory for collection variables only. To allocate memory for row variables, use the ALLOCATE ROW statement.

Examples

The following example shows how to allocate resources with the ALLOCATE COLLECTION statement for the untyped collection variable, **a_set**:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection a_set;
EXEC SQL END DECLARE SECTION;
. . .
EXEC SQL allocate collection :a_set;
```

The following example uses ALLOCATE COLLECTION to allocate resources for a typed collection variable, **a_typed_set**:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(integer not null) a_typed_set;
EXEC SQL END DECLARE SECTION;
. . .
EXEC SQL allocate collection :a_typed_set;
```

ALLOCATE DESCRIPTOR statement

Use the ALLOCATE DESCRIPTOR statement to declare the name and allocate memory for a system-descriptor area (SDA). Use this statement with ESQL/C.

Syntax

```
ALLOCATE DESCRIPTOR { 'descriptor' | descriptor_var } [ WITH MAX { items | items_var } ]
```

Element	Description	Restrictions	Syntax
descriptor	Name that you declare here for an unallocated system-descriptor area	Enclose in single (') quotation marks. Must be unique among SDA names	Quoted String on page 1202.
descriptor_var	Host variable that stores the name of a system-descriptor area	Must contain name of unallocated system-descriptor area	Language specific
items	Number of item descriptors in *descriptor*. Default value is 100.	Must be an unsigned INTEGER greater than zero	Literal Number on page 1198
items_var	Host variable that contains the number of items	Data type must be INTEGER or SMALLINT	Language specific

Usage

The ALLOCATE DESCRIPTOR statement creates a new *system-descriptor area*, which is a location in memory that holds information that the DESCRIBE statement can display, or that holds information about the WHERE clause of a query.

A system-descriptor area (SDA) contains one or more fields called *item descriptors*. Each item descriptor holds a data value that the database server can receive or send. The item descriptors also contain information about the data, such as data type, length, scale, precision, and support for NULL values.

A system-descriptor area holds information that a DESCRIBE ... USING SQL DESCRIPTOR statement obtains or that holds information about the WHERE clause of a dynamically executed query.

If the name that ALLOCATE DESCRIPTOR declares for a system-descriptor area matches the name of an existing system-descriptor area, the database server returns an error. After you free the specified descriptor with the DEALLOCATE DESCRIPTOR statement, however, the ALLOCATE DESCRIPTOR statement can reuse the same descriptor name.

WITH MAX Clause

You can use the WITH MAX clause to indicate the maximum number of item descriptors that the new system-descriptor area can include.

When you use this clause, the COUNT field value of the system-descriptor area is set to the number of *items* that you specify here. If you do not specify the WITH MAX clause, the default value of the COUNT field is `100` items. You can use the SET DESCRIPTOR statement to change the value of the COUNT field.

Examples of ALLOCATE DESCRIPTOR statements

The following examples show valid ALLOCATE DESCRIPTOR statements that include the WITH MAX clause. This example uses embedded variable names to identify the system-descriptor area and to specify the maximum number of item descriptors:

```
EXEC SQL allocate descriptor :descname with max :occ;
```

The next example uses a quoted string to declare desc1 as the identifier of the system-descriptor area, and uses an unsigned integer to specify `3` as the maximum number of item descriptors in the desc1 area:

```
EXEC SQL allocate descriptor 'desc1' with max 3;
```

ALLOCATE ROW statement

Use the ALLOCATE ROW statement to allocate memory for a *row* variable. This statement is an extension to the ANSI/ISO standard for SQL. Use this statement with ESQL/C.

Syntax

ALLOCATE ROW*variable*

Element	Description	Restrictions	Syntax
variable	Name of a typed or untyped *row* variable to allocate	Must be an unallocated row-type host variable	Language specific

Usage

The ALLOCATE ROW statement allocates memory for a host variable that stores row-type data. To create a **row** variable, an ESQL/C program must do the following:

1. Declare the *row* variable. The row variable can be a typed or untyped *row* variable.
2. Allocate memory for the *row* variable with the ALLOCATE ROW statement.

The following example shows how to allocate resources with the ALLOCATE ROW statement for the typed *row* variable, `a_row`:

```
EXEC SQL BEGIN DECLARE SECTION;
   row (a int, b int) a_row;
EXEC SQL END DECLARE SECTION;
. . .
EXEC SQL allocate row :a_row;
```

The ALLOCATE ROW statement sets **SQLCODE** (the contents of **sqlca.sqlcode**) to zero (0) if the memory allocation operation was successful, or to a negative error code if the allocation failed.

You must explicitly release memory with the DEALLOCATE ROW statement. Once you free the *row* variable with the DEALLOCATE ROW statement, you can reuse the *row* variable.

Tip: The ALLOCATE ROW statement allocates memory for the *row* variable only. To allocate memory for the *collection* variable, use the ALLOCATE COLLECTION statement.

When you use the same *row* variable in multiple function calls without deallocating it, a memory leak on the client computer results. Because there is no way to determine if a pointer is valid when it is passed, assumes that the pointer is not valid and assigns it to a new memory location.

ALTER ACCESS_METHOD statement

Use the ALTER ACCESS_METHOD statement to change one or more attributes of a user-defined primary or secondary access method in the **sysams** system catalog table.

Syntax

(explicit id unique_40_Connect_42_accmeth002) unique_40_Connect_42_accmeth002 **ALTER ACCESS_METHOD** [*owner* .] *access_method* { | { **MODIFY** | **ADD** } <Purpose Option>[] | **DROP** *purpose_keyword* }

Element	Description	Restrictions	Syntax
access_method	Name of the access method to modify	Access method must be registered in the **sysams** system catalog table by a previous CREATE ACCESS_METHOD statement	Identifier on page 1231
owner	Name of the owner of the access method	Must own the access method	Owner name on page 1262
purpose_keyword	A keyword that indicates which attribute to change	Keyword must be associated with the access method by a previous CREATE or ALTER ACCESS_METHOD statement	Purpose Functions, Flags, and Values on page 1267

Usage

This statement is an extension to the ANSI/ISO standard for SQL. This statement cannot modify a built-in access method.

Use ALTER ACCESS_METHOD to modify the definition of a user-defined access method. You cannot modify a built-in access method.

You must own the access method or hold the DBA privilege to alter a user-defined access method. In an ANSI-compliant database, the DBA must qualify the name of the access method with the *owner* name if another user is the owner of the access method.

When you alter an access method, you change the purpose-option specifications (purpose functions, purpose methods, purpose flags, or purpose values) that define the access method. For example, you might alter an access method to declare a new user-defined function or method name, or to provide a multiplier for the scan cost on a table.

If a transaction is in progress, the database server waits to modify the access method until after the transaction is committed or rolled back. No other users can execute the access method until the transaction has completed.

Examples

The following statement alters the **remote** user-defined access method:

```
ALTER ACCESS_METHOD remote
   ADD am_scancost = FS_scancost,
   ADD am_rowids,
   DROP am_getbyid,
   MODIFY am_costfactor = 0.9;
```

The preceding example makes the following changes to the access method:

- Adds a user-defined function or method named **FS_scancost( )**, which is associated in the **sysams** table with the **am_scancost** keyword
- Sets (adds) the **am_rowids** flag
- Drops the user-defined function or method associated with the **am_getbyid** keyword
- Modifies the **am_costfactor** value

ALTER FRAGMENT statement

Use the ALTER FRAGMENT statement to change the distribution strategy or the storage location of an existing table or index. This statement is an extension to the ANSI/ISO standard for the SQL language.

Syntax

(explicit id unique_44_Connect_42_altfrag002) unique_44_Connect_42_altfrag002(explicit id unique_44_Connect_42_altfrag003) unique_44_Connect_42_altfrag003(explicit id unique_44_Connect_42_altfrag004) unique_44_Connect_42_altfrag004(explicit id unique_44_Connect_42_altfrag005) unique_44_Connect_42_altfrag005(explicit id unique_44_Connect_42_altfrag006) unique_44_Connect_42_altfrag006(explicit id unique_44_Connect_42_altfrag007) unique_44_Connect_42_altfrag007(explicit id unique_44_Connect_42_altfrag008) unique_44_Connect_42_altfrag008

```
ALTER FRAGMENT [ ONLINE ] { ON TABLE surviving_table { <ATTACH Clause> | <DETACH Clause> | <INIT Clause> | <ADD Clause>
| { <DROP Clause> | <MODIFY Clause> } } | ON INDEX surviving_index { <INIT Clause> | <ADD Clause> | <DROP Clause> |
<MODIFY Clause> } }
```

Element	Description	Restrictions	Syntax
surviving_index	Index on which to modify the distribution or storage	Must exist when the statement executes	Identifier on page 1231
surviving_table	Table on which to modify the distribution or storage	Must exist. See Restrictions on the ALTER FRAGMENT Statement on page 25.	Identifier on page 1231

Usage

The ALTER FRAGMENT statement applies only to table fragments or index fragments that are located at the current site. No remote information is accessed or updated.

You must have the Alter privilege or the DBA privilege to change the fragmentation strategy of a table. You must have the Index privilege or the DBA privilege to alter the fragmentation strategy of an index.

Attention: This statement can cause indexes to be dropped and rebuilt. Before undertaking alter operations, check corresponding sections in your to review effects and strategies.

Clauses of the ALTER FRAGMENT statement support the following tasks.

Clause

Effect

ATTACH

Combines two or more tables that have the same schema into a single fragmented table

DETACH

Detaches one fragment from a fragmented table, and creates a new nonfragmented table to store the rows in the fragment.

INIT

Provides the following options:

- Defines and initializes a fragmentation strategy on a nonfragmented table
- Changes the order of evaluation of fragment expressions
- Changes the fragmentation strategy of a fragmented table or index
- Changes the storage location of an existing table
- Moves data from an existing table fragment into a new nonfragmented table
- Changes the storage location of fragments that the database generates for a table or index
- Changes the fragmentation key or fragmentation expression for a table or index

ADD

Adds an additional fragment to an existing fragmentation list

DROP

Drops an existing fragment from a fragmentation list

Remove one or more dbspaces from the list of dbspaces where interval fragments are created.

MODIFY

Changes an existing interval, list, or expression-based fragmentation expression

Moves an existing fragment to a different dbspace

Replaces with a new list the current list of dbspaces where interval fragments are created.

Enables or disables automatic creation of interval fragments

Use the CREATE TABLE statement or the INIT clause of the ALTER FRAGMENT statement to create fragmented tables.

After a dbspace has been renamed successfully by the **onspaces** utility, only the new name can reference the renamed dbspace. The database server automatically updates existing fragmentation strategies for tables or indexes in the system catalog, however, to replace the old dbspace name with the new name. You do not need to take any additional action to update a distribution strategy or storage location that was defined using the old dbspace name, but you must use the new name if you reference the dbspace in an ALTER FRAGMENT or ALTER TABLE statement.

If you omit the optional ONLINE keyword, the ALTER FRAGMENT operation requires exclusive access and exclusive locks on all of the tables involved in the operation. If you enable the FORCE_DDL_EXEC session environment option, you can force out other transactions that have opened a table involved in an ALTER FRAGMENT ON TABLE operation, or that have placed locks on any of those tables. If the server is unable to get exclusive access and exclusive locks on the table, the server starts rolling back the transactions that are open or that have locks on the table, until the value specified with the FORCE_DDL_EXEC option is reached. (For more information, see FORCE_DDL_EXEC session environment option.)

Restrictions on the ALTER FRAGMENT Statement

You cannot use the ALTER FRAGMENT statement on a view, on a temporary table, or on a table that is not registered in the current database.

If your table or index is not already fragmented, the only clauses available to you are ATTACH and INIT.

You cannot use ALTER FRAGMENT on a typed table that is part of a table hierarchy.

ALTER FRAGMENT and Transaction Logging

If your database supports transaction logging, ALTER FRAGMENT is executed within a single transaction. If the fragmentation strategy uses large numbers of records, you might run out of log space or disk space. (To alter a fragmentation strategy, the database server requires extra disk space that it later frees.)

If you run out of log space or disk space, try one of the following procedures to reduce your log-space or disk-space requirements:

- Turn off logging and turn it back on again at the end of the operation. This procedure indirectly requires a backup of the **root** dbspace.
- Split the operations into multiple ALTER FRAGMENT statements, moving a smaller portion of records each time.

For information about log-space requirements and disk-space requirements, see your *HCL® Informix® Administrator's Guide*. That guide also contains detailed instructions about how to turn off logging. For information about backups, refer to your *HCL® Informix® Backup and Restore Guide*.

Determining the Number of Rows in the Fragment

About this task

You can place as many rows into a fragment as the available space in the dbspace allows.

To find out how many rows are in a fragment:

1. Run the UPDATE STATISTICS FORCED statement on the table. This step fills the **sysfragments** system catalog table with the current table information.
2. Query the **sysfragments** system catalog table to examine the **npused** and **nrows** values. The **npused** column shows the number of data pages used in the fragment, and the **nrows** column shows the number of rows in the fragment.

The ONLINE keyword in ALTER FRAGMENT operations

The ONLINE keyword instructs the database server to modify the storage of the table in the background, while other concurrent users can continue to access the table.

The DBA can reduce the risk of nonexclusive access errors, and can increase the availability of the fragmented table, by including the ONLINE keyword in the ALTER FRAGMENT statement. This instructs the database server to commit the work in ATTACH, DETACH, and MODIFY operations internally, if there are no errors, and to apply an intent exclusive lock to the table, rather than an exclusive lock.

In DETACH and MODIFY operations, the ONLINE keyword can reduce the risk of -710 errors when either of the following is true:

- the AUTO_REPREPARE configuration parameter is set to `1`,
- the IFX_ AUTO_REPREPARE session environment variable is set to `1`.

The following restrictions apply to the ALTER FRAGMENT ONLINE FOR TABLE statement:

- Only the ATTACH, DETACH, and MODIFY options to ALTER FRAGMENT ONLINE are valid.
- The FOR TABLE clause must specify a table that is fragmented by a range interval scheme.
- The table that is being altered cannot be locked explicitly by the LOCK TABLE statement.
- The ALTER FRAGMENT ONLINE operation must be the first statement in the transaction that modifies any database object or table.
- No other operation that modifies an object in the database can follow the ALTER FRAGMENT ONLINE statement in the same transaction.

For additional information, see the following topics:

- Using the ONLINE keyword in ATTACH operations on page 31
- Using the ONLINE keyword in DETACH operations on page 38
- Using the ONLINE keyword in MODIFY operations on page 67

Automatic renaming of interval fragment identifiers

Some ALTER FRAGMENT operations can change the positional order of existing interval fragments within the fragment list. In these cases, the database server automatically updates the system-defined names of the affected interval fragments.

For tables partitioned by an interval fragmentation scheme, ALTER FRAGMENT operations that add, drop, attach, or detach fragments, or that modify the transition value of the table, can change the **sysfragments.evalpos** values for existing interval

fragments, or can change an interval fragment to a range fragment. To avoid creating new interval fragments with the same system-generated name as an interval fragment that the ALTER FRAGMENT statement has repositioned within the fragment list, the database server automatically replaces the original system-defined names with new identifiers that will not match the names of subsequently created interval fragments of the same table.

The general rules used for system generated range and interval fragment names are as follows:

- For interval fragments: `sys_evalpos`
- For range fragments: `sys_evalposrg`

Here `evalpos` is the numeric (ordinal) value of **sysfragments.evalpos**, where `0` is the `evalpos` value for the first fragment in the fragment list.

During a fragment renaming operation, an exclusive lock is placed on the fragment while the **sysfragments** system catalog is being updated with the new **partition** names, and with new **evalpos** values for any fragments whose ordinal positions within the fragment list changed during the ALTER FRAGMENT operation.

To avoid declaring non-unique fragment names when new interval fragments are created, the database server renames only system-generated identifiers of interval fragment that are repositioned during ALTER FRAGMENT operations. Automatic renaming does not occur for user-defined identifiers of repositioned fragments.

If you wish to avoid having existing fragments automatically renamed during ALTER FRAGMENT ONLINE ATTACH statements, or during other ALTER FRAGMENT operations on tables that use interval partitioning, you can first use the ALTER FRAGMENT MODIFY statement to rename with user-defined names the interval fragments whose system-generated names might otherwise be changed by the ALTER FRAGMENT operation. User-defined fragment names cannot begin with the string `sys_`.

ATTACH Clause

Use the ATTACH clause of the ALTER FRAGMENT ON TABLE statement to combine tables that have identical structures into a table with the same fragmentation strategy.

You can use this syntax, for example, to combine a fragment that has been detached from a table with an archival table that has the same distributed storage scheme.

(explicit id unique_45_Connect_42_attach002) unique_45_Connect_42_attach002(explicit id unique_45_Connect_42_attach003)

unique_45_Connect_42_attach003(explicit id unique_45_Connect_42_attach004) unique_45_Connect_42_attach004

ATTACH Clause

"**ATTACH**"

"{[*surviving_table* | *consumed_table*} [<AS Clause>]"

AS Clause

"**AS** [[**PARTITION***new_frag*]"

"{{ *expr* <List Expression> } [[{ **AFTER** | **BEFORE** }] *old_frag*] | { <Range Interval Expression> | [[**REMAINDER** } }"

Range Interval Expression

"{ **VALUES** < *range_expr* | [**VALUES IS NULL** }"

List Expression

"**VALUES** {({ *const_expr* }) | [**[IS] NULL** }"

Element	Description	Restrictions	Syntax
const_expr	Constant expression that defines the list of values for a fragment to store	Must be a quoted string or a literal value. Each value in the list must be unique among the lists for fragments of the same object.	Constant Expressions on page 1040
consumed _table	Table that loses its identity, to be merged with *surviving_table*	Schema must match that of *surviving _table*. Cannot include serial columns, nor unique, referential, or primary key constraints. See also General Restrictions for the ATTACH Clause on page 29.	Identifier on page 1231
expr	Expression defining which rows are stored in a fragment of a table partitioned by expression	Can include only columns from the current table and only data values from a single row. See also General Restrictions for the ATTACH Clause on page 29.	Condition on page 967; Expression on page 1003

Element	Description	Restrictions	Syntax
new_frag	Name declared here for a *consumed_table* fragment. Default is the dbspace name.	Must be unique among the names of fragments of *surviving_table*	Identifier on page 1231
old_frag	Fragment or dbspace name for a *surviving_table* fragment	Must exist. Cannot be a range or interval fragment.	Identifier on page 1231
range _expr	Constant expression that defines the upper bound for fragment key values stored in the fragment	Must be a constant literal expression that evaluates to a numeric, DATETIME, or DATE data type compatible with the data type of the fragment key expression	Constant Expressions on page 1040
surviving _table	Table on which to modify the distribution or storage location	Must exist. Cannot have any constraints. See also Restrictions on the ALTER FRAGMENT Statement on page 25.	Identifier on page 1231

When a new expression fragment is attached to table that is fragmented by list or by range interval, the data from the consumed table and the affected fragments in the surviving table are scanned and moved into appropriate partitions, because these strategies are not overlapping.

If the automatic mode for updating distribution statistics is enabled, and the table being attached to has fragmented distribution statistics, the database server calculates the distribution statistics of the new fragment. Stale distribution statistics of existing fragments are also recalculated at this point. This recalculation of fragment statistics runs in the background. After the database server has calculated the fragment statistics, it merges them to form table distribution statistics, and stores the results in the system catalog.

To use this clause, you must have the DBA privilege or else be the owner of the specified tables. The ATTACH clause supports the following tasks:

- Creates a single fragmented table by combining two or more identically-structured, nonfragmented tables

  (See Combining Nonfragmented Tables to Create a Fragmented Table on page 31.)
- Attaches one or more tables to a fragmented table

  (See Attaching a Table to a Fragmented Table on page 31.)

General Restrictions for the ATTACH Clause

This clause is not valid in ALTER FRAGMENT ON INDEX statements.

Any tables that you attach must have been created previously in separate partitions. You cannot attach the same table more than once.

All consumed tables listed in the ATTACH clause must have the same structure as the surviving table. The number, names, data types, and relative position of the columns must be identical.

The *expression* cannot include aggregates, subqueries, or variant functions.

Additional Restrictions on the ATTACH Clause

User-defined routines and references to fields of a ROW-type column are not valid. You cannot attach a fragmented table to another fragmented table.

All of the dbspaces that store the fragments must have the same page size.

An ATTACH operation on two nonfragmented tables cannot produce a surviving table that is fragmented by interval or by list. (If you want to attach two nonfragmented tables, use the INIT option of ALTER FRAGMENT to define an interval or list fragmentation scheme for one of the nonfragmented tables, and then use the ATTACH option to attach the second table to it.)

For surviving tables that are fragmented by interval, the following restrictions apply:

- Because the database server determines the ordinal position of interval fragments, BEFORE and AFTER specifications are not valid.
- You cannot attach a fragment whose expression matches an existing interval fragment expression.
- While you attach fragments above the transition value, the upper limit of the fragment being attached must align at an interval fragment boundary. That is, the upper limit of the fragment must equal the transition value plus an integer multiple of the interval value.

For fragmented tables that are protected by a security policy, attaching a fragment to the table fails if any of the following conditions are not satisfied:

- The source table and the target table are both protected by the same security policy;
- Both tables have the same protection granularity (either row-level, or column-level, or both row-level and column-level);
- In both tables, the same set of protected columns is protected by the same security labels. If there is more than one protected column, there can be more than one security label in each table, but the same label must protect the same column in both tables.

If the ATTACH operation fails because one or more of these conditions are not satisfied, you can use the ALTER TABLE statement to make the schemas of the two tables identical, and then repeat the ALTER FRAGMENT ATTACH statement on the modified tables

Only a user who holds the DBSECADM role can reference a protected table in the ALTER FRAGMENT statement.

Using the BEFORE, AFTER, and REMAINDER options

The BEFORE and AFTER options allow you to place a new fragment either before or after an existing fragment. You cannot use the BEFORE and AFTER options when the distribution scheme is round-robin or range interval..

When you attach a new list or expression fragment without specifying an explicit BEFORE or AFTER keyword option, the database server places the added fragment at the end of the fragment list, unless a remainder fragment exists. If a

remainder fragment exists, the new fragment, by default, is placed just before the remainder fragment. You cannot attach a new fragment after the remainder fragment.

You cannot define a remainder fragment when the distribution scheme is round-robin or range interval.

If you omit the AS PARTITION *fragment* specification, the default name of the fragment is the name of the dbspace where it is stored. If another fragment of the same table already has the same name as the dbspace, the database server issues an exception, and the ALTER FRAGMENT ATTACH operation fails.

Combining Nonfragmented Tables to Create a Fragmented Table

When you transform tables with identical table structures into fragments into a single table, you allow the database server to manage the fragmentation instead of allowing the application to manage the fragmentation. The distribution scheme can be round-robin or expression based.

To make a single, fragmented table from two or more identically-structured, nonfragmented tables, the ATTACH clause must include the surviving table in the *attach list*. The attach list is the specified list of tables in the ATTACH clause.

To include a **rowid** column in the newly-created single, fragmented table, attach all tables first and then add the **rowid** with the ALTER TABLE statement.

Attaching a Table to a Fragmented Table

About this task

To attach a nonfragmented table to an already fragmented table, the nonfragmented table must have been created in a separate dbspace and must have the same table structure as the fragmented table. In the following example, a round-robin distribution scheme fragments the table **cur_acct**, and the table **old_acct** is a nonfragmented table that resides in a separate dbspace. The following example shows how to attach (as the consumed table) **old_acct** to **cur_acct** (as the surviving table):

```
ALTER FRAGMENT ON TABLE cur_acct ATTACH old_acct;
```

When you attach one or more tables to a fragmented table, a *consumed_table* must be nonfragmented.

Using the ONLINE keyword in ATTACH operations

The ONLINE keyword instructs the database server to commit the ALTER FRAGMENT ATTACH work internally, if there are no errors, and to apply an intent exclusive lock to the surviving table, rather than an exclusive lock. An exclusive lock is applied to the consumed table, which must be a nonfragmented table.

Requirements for ONLINE ATTACH operations

You can use the ATTACH option to the ALTER FRAGMENT ONLINE ON TABLE statement only if the *surviving table* is fragmented by an interval fragmentation scheme. The consumed table must be nonfragmented.

All indexes on the surviving table must have the same fragmentation scheme as the table. (That is, any indexes must be attached.) For this reason, if there is a primary key constraint or other referential constraints on the table, it is recommended

that you first create an attached index for the constraint, and then use the ALTER TABLE statement to add the constraint. (By default, system-created indexes for primary key constraints and for other referential constraints are detached.)

For each index on the surviving table, there must be a matching index on the same set of columns of the consumed table. The matching indexes on the consumed table will be recycled as index fragments on the surviving table during the ATTACH operation. Any additional indexes on the consumed table are dropped during the ATTACH operation. The indexes on the consumed table that will be recycled must each be detached in a single dbspace, and the dbspace that stores the recycled index must be same dbspace that stores the consumed table.

If the index on the surviving table is unique, the corresponding matching index on the consumed table must also be unique.

The consumed table must have a check constraint that satisfies the following two conditions:

- It must exactly match the expression for the fragment that is being attached.
- It must span a single interval only.

This last requirement, that rows in the consumed table span only a single interval within the range interval fragmentation scheme of the surviving table, is necessary to prevent data movement. Data movement is not allowed in ALTER FRAGMENT ATTACH operations that include the ONLINE keyword.

Only one consumed table can be specified in the ONLINE ATTACH operation.

All other restrictions that apply to the ATTACH option also apply to ONLINE ATTACH operations. For those restrictions, see General Restrictions for the ATTACH Clause on page 29 and Additional Restrictions on the ATTACH Clause on page 30.

Example of ALTER FRAGMENT ONLINE ATTACH

The following SQL statements define a fragmented **employee** table that uses a range-interval storage distribution scheme, with a unique index **employee_id_idx** on the column **emp_id** (that is also the fragmentation key) and another index **employee_dept_idx** on the column **dept_id**.

```
CREATE TABLE employee
     (emp_id INTEGER, name CHAR(32),
      dept_id CHAR(2), mgr_id INTEGER, ssn CHAR(12))
  FRAGMENT BY RANGE (emp_id)
    INTERVAL (100) STORE IN (dbs1, dbs2, dbs3, dbs4)
      PARTITION p0 VALUES < 200 IN dbs1,
      PARTITION p1 VALUES < 400 IN dbs2;
CREATE UNIQUE INDEX employee_id_idx ON employee(emp_id);
CREATE INDEX employee_dept_idx ON employee(dept_id);
```

The last two statements insert rows with fragment key values above the upper limit of the transition fragment, causing the database server to generate two new interval fragments, so that the resulting fragment list consists of four fragments:

```
Fragments in surviving table before ALTER FRAGMENT ONLINE:
p0     VALUES < 200                   - range fragment
p1     VALUES < 400                   - range fragment (transition fragment)
sys_p2 VALUES >= 400 AND VALUES < 500 - interval fragment
sys_p4 VALUES >= 600 AND VALUES < 700 - interval fragment
```

The next SQL statements define a nonfragmented **employee2** table with the same column schema as the **employee** table, and with single-column indexes on the two corresponding columns (**emp_id** and **dept_id**) that were indexed in the **employee** table, but also defines a unique index **employee2_ssn_idx** on the column **emp_ssn** and another index **employee_dept_idx** on the column **name**. All four of these indexes are stored in the dbspace **dbs4**. The CREATE TABLE statement also specifies a check constraint (`(emp_id >=500 AND emp_id <600)`) that exactly matches the fragment expression for a consumed table that will be attached, and that exactly spans a single interval of the range interval fragmentation scheme for the **employee** table.

```
CREATE TABLE employee2
     (emp_id INTEGER, name CHAR(32),
      dept_id  CHAR(2), mgr_id INTEGER, ssn CHAR(12),
  CHECK (emp_id >=500 AND emp_id <600)) in dbs4;
CREATE UNIQUE INDEX employee2_id_idx ON employee2(emp_id) in dbs4;
CREATE INDEX employee2_dept_idx ON employee2(dept_id) in dbs4;
CREATE UNIQUE INDEX employee2_ssn_idx ON employee2(ssn) in dbs4;
CREATE INDEX employee2_name_idx ON employee2(name) in dbs4;
```

The following statement returns an error because the fragment being attached is a range fragment (a fragment that stores rows with fragmentation key values below the transition value of 400 for the **employee** table). Only interval fragments can be attached online.

```
ALTER FRAGMENT ONLINE ON TABLE employee
   ATTACH employee2 AS PARTITION p3 VALUES < 300;
```

The following statement runs successfully, and creates a new **p3** interval fragment:

```
ALTER FRAGMENT ONLINE ON TABLE employee
   ATTACH employee2 AS PARTITION p3 VALUES < 600;

Fragments in surviving table after ALTER FRAGMENT ONLINE:
p0     VALUES < 200                   - range fragment
p1     VALUES < 400                   - range fragment
sys_p2 VALUES >= 400 AND VALUES < 500 - interval fragment
sys_p3 VALUES >= 500 AND VALUES < 600 - interval fragment
sys_p4 VALUES >= 600 AND VALUES < 700 - interval fragment
```

Note that the successful ALTER FRAGMENT ONLINE . . . ATTACH operation above required multiple correspondences among specifications in the DDL statements that defined the surviving and consumed tables, including their columns, indexes, constraints, index storage location, and the interval fragmentation strategy of the surviving table:

- The check constraint on the consumed table spans a single interval only. The interval value is 100 for the surviving table, and the check constraint is `>= 500 and < 600`.
- The conditional expression being attached (`< 600`) is internally converted to the interval fragment expression format (`>= 500 and < 600`) which matches the check constraint.
- The indexes on the surviving table are attached (that is, they are fragmented by the same fragmentation scheme as the table) because no fragmentation strategy was specified explicitly in their CREATE INDEX statements.
- The indexes on the consumed table are detached in a single dbspace (**dbs4**), which is the same dbspace that stored the consumed table.
- For each index on the surviving table, there is a matching index on the consumed table.
- The extra indexes on the consumed table (**employee2_ssn_idx** and **employee2_name_idx**) that do not correspond to indexes on the surviving **employee** table are dropped during the ONLINE ATTACH operation.

Effect of the ATTACH Clause

After an ATTACH operation, all consumed tables no longer exist. Any CHECK constraints or NOT NULL constraints on the consumed tables also no longer exist. You must reference the records that were in the consumed tables through the surviving table.

What Happens to Indexes?

A detached index on the surviving table retains its same fragmentation strategy. That is, a detached index does not automatically adjust to accommodate the new fragmentation of the surviving table. For more information on what happens to indexes, see the discussion about altering table fragments in your .

In a database that supports transaction logging, an ATTACH operation extends any attached index on the surviving table according to the new fragmentation strategy of the surviving table. All rows in the consumed table are subject to these automatically adjusted indexes. For information on whether the database server completely rebuilds the index on the surviving table or reuses an index that was on the consumed table, see your .

In a nonlogging database of Informix®, an ATTACH operation does not extend indexes on the surviving table according to the new fragmentation strategy of the surviving table. To extend the fragmentation strategy of an attached index according to the new fragmentation strategy of the surviving table, you must drop the index and re-create it on the surviving table.

Some ALTER FRAGMENT ... ATTACH operations to attach a fragment can cause the database server to update the index structure. When an index is rebuilt in those cases, the associated column distribution is automatically recalculated, and is available to the query optimizer when it designs query plans for the table on which the fragment was attached.

- For an indexed column (or a set of columns) on which ALTER FRAGMENT ... ATTACH automatically rebuilds a B-tree index, the recalculated column distribution statistics are equivalent to distributions created by the UPDATE STATISTICS statement in HIGH mode.
- If the rebuilt index is not a B-tree index, the automatically recalculated statistics correspond to distributions created by the UPDATE STATISTICS statement in LOW mode.

See also the section Automatic Calculation of Distribution Statistics on page 261 in the description of the CREATE INDEX statement for additional information about statistical distributions that are produced automatically when an index or constraint is created on an existing table.

What Happens to BYTE and TEXT Columns?

When an ATTACH occurs, BYTE and TEXT fragments of the consumed table become part of the surviving table and continue to be associated with the same rows and data fragments as they were before the ATTACH operation.

Each BYTE and TEXT column in every table that is specified in the ATTACH clause must have the same storage type, either blobspace or tblspace. If the BYTE or TEXT column is stored in a blobspace, the same column in all tables must be in the same blobspace. If the BYTE or TEXT column is stored in a tblspace, the same column must be stored in a tblspace in all tables.

What Happens to Triggers and Views?

When you attach tables, triggers on the surviving table survive the ATTACH, but triggers on the consumed table are automatically dropped. No triggers are activated by the ATTACH clause, but subsequent data-manipulation operations on the new rows can activate triggers.

Views on the surviving table survive the ATTACH operation, but views on the consumed table are automatically dropped.

What Happens with the Distribution Scheme?

You can attach a nonfragmented table to a table with any type of supported distribution scheme. In general, the resulting table has the same fragmentation strategy as the prior fragmentation strategy of the surviving table.

When you attach two or more nonfragmented tables, however, the distribution scheme can either be based on expression or round-robin.

Only the following distribution schemes can result from combining the distribution schemes of the tables in the ATTACH clause.

Prior Distribution Scheme of Surviving Table	Prior Distribution Scheme of Consumed Table	Resulting Distribution Scheme
None	None	Round-robin or expression
Round-robin	None	Round-robin
Expression	None	Expression

Round-Robin Distribution Scheme

The following example combines nonfragmented tables **pen_types** and **pen_makers** into a single, fragmented table, **pen_types**. Table **pen_types** resides in dbspace **dbsp1,** and table **pen_makers** resides in dbspace **dbsp2**. Table structures are identical in each table.

```
ALTER FRAGMENT ON TABLE pen_types ATTACH pen_types, pen_makers;
```

After you execute the ATTACH clause, the database server fragments the table **pen_types** using a round-robin scheme into two dbspaces: the dbspace that contained **pen_types** and the dbspace that contained **pen_makers**. Table **pen_makers** is consumed and no longer exists; all rows that were in table **pen_makers** are now in table **pen_types**.

Expression Distribution Scheme

Consider the following example that combines tables **cur_acct** and **new_acct** and uses an expression-based distribution scheme. Table **cur_acct** was originally created as a fragmented table and has fragments in dbspaces **dbsp1** and **dbsp2**. The first statement of the example shows that table **cur_acct** was created with an expression-based distribution scheme. The second statement of the example creates table **new_acct** in **dbsp3** without a fragmentation strategy. The third statement combines the tables **cur_acct** and **new_acct**. Table structures (columns) are identical in each table.

```
CREATE TABLE cur_acct (a int) FRAGMENT BY EXPRESSION
   a < 5 in dbsp1, a >= 5 and a < 10 in dbsp2;
CREATE TABLE new_acct (a int) IN dbsp3;
ALTER FRAGMENT ON TABLE cur_acct ATTACH new_acct AS a>=10;
```

When you examine the **sysfragments** system catalog table after you alter the fragment, you see that table **cur_acct** is fragmented by expression into three dbspaces. For additional information about the **sysfragments** system catalog table, see the *HCL® Informix® Guide to SQL: Reference*.

In addition to simple range rules, you can use the ATTACH clause to fragment by expression with hash or arbitrary rules. For a discussion of all types of expressions that you can use in an expression-based distribution scheme, see Expression Fragment Clause on page 358.

Warning: When you specify a date value as the default value for a parameter, make sure to specify 4 digits instead of 2 digits for the year. If you specify a 2-digit year, the setting of the **DBCENTURY** environment variable can affect how the database server interprets the date value. For more information, see the *HCL® Informix® Guide to SQL: Reference*.

DETACH Clause

Use the DETACH clause of the ALTER FRAGMENT ON TABLE statement to detach a table fragment from a storage distribution scheme and place the contents into a new nonfragmented table.

This clause is not valid in ALTER FRAGMENT ON INDEX statements.

For an explanation of distribution schemes, see FRAGMENT BY clause on page 356.

DETACH Clause

" **DETACH** [**PARTITION**] *fragmentnew_table* "

Element	Description	Restrictions	Syntax
fragment	Named fragment or dbspace that contains the table fragment to be detached.	Must exist at the time of execution. For list or range- interval fragments, the PARTITION keyword must precede *fragment*.	Identifier on page 1231

Element	Description	Restrictions	Syntax
new_table	Name that you declare here for a nonfragmented table that results after you execute the ALTER FRAGMENT statement.	Must not exist before execution	Identifier on page 1231

Usage

The new table that results from executing the DETACH clause does not inherit any indexes or constraints from the original table. Only data values remain.

Similarly, the new table does not inherit any access privileges from the original table. Instead, the new table has the default privileges of any new table. For further information on default table-level privileges, see the description in the GRANT statement of Table-Level Privileges on page 540.

The DETACH clause cannot be applied to a table that has any of the following attributes:

- a ROWID column
- a column or columns defined as the primary key of a referential constraint
- an Enterprise Replication replicate is defined on the table
- a detached index (that is, an index whose storage distribution scheme is not identical to the fragmentation strategy of the table).

If you omit the PARTITION keyword, the name of the fragment must be the name of the dbspace where the fragment is stored.

In the following example, a system-generated range interval fragment **sys_pt1** is detached from table **T1** and placed in a new unfragmented table **detacht1**:

```
ALTER FRAGMENT ON TABLE T1 DETACH PARTITION sys_pt1 detacht1;
```

The next example detaches a list fragment **part2** from table **T2** and places its data in a new unfragmented table **detacht2**:

```
ALTER FRAGMENT ON TABLE T2 DETACH PARTITION part2 detacht2;
```

Distribution statistics after DETACH operations

Some ALTER FRAGMENT . . . DETACH operations can cause the database server to update the index structure of the original table. When an index is rebuilt in those cases, the database server also recalculates the associated column distributions, and these statistics are available to the query optimizer when it designs query plans for the table from which the fragment was detached:

- For an indexed column (or for a set of columns) on which ALTER FRAGMENT . . . DETACH automatically rebuilds a B-tree index, the recalculated column distribution statistics are equivalent to distributions created by the UPDATE STATISTICS statement in HIGH mode.
- If the rebuilt index is not a B-tree index, the automatically recalculated statistics correspond to distributions created by the UPDATE STATISTICS statement in LOW mode.

If the automatic mode for updating column-distribution statistics is enabled, and the table from which the fragment is being detached has fragment-level distribution statistics, the database server uses the statistics of the detached fragment as distribution statistics for the new table. The database server also merges the data distribution statistics of the remaining fragments to calculate new table distribution statistics for the original table, and stores these results in the **sysdistrib** system catalog table. This registration of distribution statistics for the new table and recalculation of table distribution statistics for the old table runs in the background.

See also the section Automatic Calculation of Distribution Statistics on page 261 in the description of the CREATE INDEX statement for additional information about statistical distributions that are produced automatically when an index or constraint is created on an existing table.

Using the ONLINE keyword in DETACH operations

The ONLINE keyword instructs the database server to commit the ALTER FRAGMENT ... DETACH work internally, if there are no errors, and to apply an intent exclusive lock to the table from which the fragment was detached, rather than an exclusive lock. An exclusive lock is applied to the table that is created from the detached fragment.

You can use the DETACH option to the ALTER FRAGMENT ONLINE ON TABLE statement only for a table that uses a range interval fragmentation scheme.

A table that uses a range interval storage distribution scheme can have two types of fragments:

- *range* fragments, which are defined by the user in the FRAGMENT BY or PARTITION BY clause of the CREATE TABLE or ALTER TABLE statement, and
- *interval* fragments, which are generated automatically by the database server during INSERT and UPDATE operations, if a row has a fragment key values above the upper limit of the transition fragment (the last range fragment).

Only an interval fragment can be detached in an ONLINE DETACH operation.

If the detached interval fragment that is not the last fragment, the database server modifies the fragment names for any system-generated interval fragments that follow the detached fragment in the fragment list to match their new **sysfragments.evalpos** values in the surviving table. During this fragment renaming operation, an exclusive lock is placed on the fragments while the **sysfragments** system catalog is being updated with the new **partition** names (and with new **evalpos** values for any fragments whose ordinal positions within the fragment list changed during the ALTER FRAGMENT DETACH operation).

All indexes on the surviving table must have the same fragmentation scheme as the table. (That is, any indexes must be attached.) For this reason, if there is a primary key constraint or other referential constraints on the table, it is recommended that you first create an attached index for the constraint, and then use the ALTER TABLE statement to add the constraint. (By default, system-created indexes for primary key constraint and for other referential constraints are detached.)

If there are sessions accessing the same partition that is being detached, it is recommended that you issue the SET LOCK MODE TO WAIT statement for enough seconds to prevent nonexclusive access errors.

All other restrictions that apply to the DETACH option also apply to ONLINE DETACH operations. For those restrictions, see Restrictions on the ALTER FRAGMENT Statement on page 25 and DROP Clause on page 50.

Example of ALTER FRAGMENT ONLINE ... DETACH

The following SQL statements define a fragmented **employee** table that uses a range-interval storage distribution scheme, with a unique index **employee_id_idx** on the column **emp_id** (that is also the fragmentation key) and another index **employee_dept_idx** on the column **dept_id**.

```
CREATE TABLE employee (emp_id INTEGER, name CHAR(32), dept_id CHAR(2),
             mgr_id INTEGER, ssn CHAR(12))
  FRAGMENT BY RANGE (emp_id)
    INTERVAL (100) STORE IN (dbs1, dbs2, dbs3, dbs4)
      PARTITION p0 VALUES < 200 IN dbs1,
      PARTITION p1 VALUES < 400 IN dbs2;
CREATE UNIQUE INDEX employee_id_idx ON employee(emp_id);
CREATE INDEX employee_dept_idx ON employee(dept_id);

INSERT INTO employee VALUES (401, "Susan", "DV", 101, "123-45-6789");
INSERT INTO employee VALUES (601, "David", "QA", 104, "987-65-4321");
```

The last two statements insert rows with fragment key values above the upper limit of the transition fragment, causing the database server to generate two new interval fragments, so that the resulting fragment list consists of four fragments:

```
Fragments in surviving table before ALTER FRAGMENT ONLINE:
p0     VALUES < 200                   - range fragment
p1     VALUES < 400                   - range fragment (transition fragment)
sys_p2 VALUES >= 400 AND VALUES < 500 - interval fragment
sys_p4 VALUES >= 600 AND VALUES < 700 - interval fragment
```

The following statement returns an error, because the specified fragment to be detached is a range fragment (a fragment that stores rows with fragmentation key values below the transition value of 400). Only interval fragments can be detached online.

```
ALTER FRAGMENT ONLINE ON TABLE employee
   DETACH PARTITION p0 employee3;
```

The following statement runs successfully, and creates a new **employee3** table to store the data in the detached fragment.

```
ALTER FRAGMENT ONLINE ON TABLE employee
   DETACH PARTITION sys_p2 employee3;
```

If there are concurrent sessions accessing **sys_p2**, set the lock mode to WAIT (for a number of seconds sufficient for the ONLINE DETACH operation to be committed) to prevent nonexclusive access errors:

```
SET LOCK MODE TO WAIT 300;
ALTER FRAGMENT ONLINE ON TABLE employee DETACH PARTITION sys_p2 employee3;

Fragments in surviving table after ALTER FRAGMENT ONLINE:
p0     VALUES < 200                   - range fragment
p1     VALUES < 400                   - range fragment
sys_p4 VALUES >= 600 AND VALUES < 700 - interval fragment.
```

Detach with BYTE and TEXT Columns

If the DETACH clause specifies the first fragment of a table that includes simple large objects of data type BYTE or TEXT, the database server applies locks on the blobspaces of every fragment of the table. To detach any other fragment of the table

locks only the blobspaces of the specified fragment, rather than the blobspaces of all fragments, so fewer locks are required if the fragment is not the first.

Detach from a Protected Table

If a successfully executed DETACH clause specifies a table that is protected by a security policy, the database server creates a new table that is protected by the same security policy.

If the original table has both row-level and column-level protection,

- the new table has the same IDSSECURITYLABEL column, and the same label, for row-level security,
- and the same set of protected columns secured by the same labels as the original table for column-level protection.

For tables with row-level protection, the IDSSECURITYLABEL column has a NOT NULL constraint by default. Only a user who holds the DBSECADM role can reference a protected table in the ALTER FRAGMENT statement.

Detach That Results in a Nonfragmented Table

The following example uses the table **cur_acct** fragmented into two dbspaces, **dbsp1** and **dbsp2**:

```
ALTER FRAGMENT ON TABLE cur_acct DETACH dbsp2 accounts;
```

This example detaches **dbsp2** from the distribution scheme for **cur_acct** and places the rows in a new table, **accounts**. Table **accounts** now has the same structure (column names, number of columns, data types, and so on) as table **cur_acct**, but the table **accounts** does not contain any indexes or constraints from the table **cur_acct**. Both tables are now nonfragmented. The following example shows a table that contains three fragments:

```
ALTER FRAGMENT ON TABLE bus_acct DETACH dbsp3 cli_acct;
```

This statement detaches **dbsp3** from the distribution scheme for **bus_acct** and places the rows in a new table, **cli_acct**. Table **cli_acct** now has the same structure (column names, number of columns, data types, and so on) as **bus_acct,** but the table **cli_acct** does not contain any indexes or constraints from the table **bus_acct**. Table **cli_acct** is a nonfragmented table, but table **bus_acct** remains fragmented.

INIT Clause

The INIT clause of the ALTER FRAGMENT statement can define or modify the fragmentation strategy or the storage location of an existing table or an existing index.

The INIT clause has the following syntax.

(explicit id unique_47_Connect_42_init002) unique_47_Connect_42_init002(explicit id unique_47_Connect_42_init003) unique_47_Connect_42_init003

INIT Clause

"INIT"

"[WITH ROWIDS]"

"{ <FRAGMENT BY Clause for Tables> | [PARTITION*fragment*] IN *dbspace* | <FRAGMENT BY Clause for Indexes> }"

Element	Description	Restrictions	Syntax
dbspace	Dbspace storing fragmented data	Must exist at time of execution	Identifier on page 1231
fragment	Name of fragment	No more than 2048 for same table	Identifier on page 1231

The INIT clause can accomplish tasks that include the following:

- Move a nonfragmented table from one dbspace to a named fragment or to another dbspace.
- Convert a fragmented table to a nonfragmented table.
- Fragment an existing non-fragmented table without redefining it.
- Convert a fragmentation strategy to another fragmentation strategy.
- Fragment an existing index that is not fragmented without redefining the index.
- Convert a fragmented index to a nonfragmented index.
- Add a new **rowid** column to the table definition.

When you use the INIT clause to modify a table, the **tabid** value in the system catalog tables changes for the affected table. The **constrid** value of all unique and referential constraints on the table also changes.

For more information about the storage spaces in which you can store a table, see Using the IN Clause on page 351.

Attention: When you execute the ALTER FRAGMENT statement with this clause, it results in data movement if the table contains any data. If data values move, the potential exists for significant logging, for the transaction being aborted as a long transaction, and for a relatively long exclusive lock being held on the affected tables. Use this statement when it does not interfere with day-to-day operations.

WITH ROWIDS Option

Nonfragmented tables contain a hidden column called **rowid**. Its integer value defines the physical location of a row. To include a **rowid** column in a fragmented table, you must explicitly request it by using the WITH ROWIDS clause in CREATE TABLE (or ADD ROWIDS in ALTER TABLE, or WITH ROWIDS in ALTER FRAGMENT INIT).

The **rowid** in a row of a fragmented table does not identify a physical location for the row in the same way that a **rowid** in a non-fragmented table does.

When you use the WITH ROWIDS option to add a new **rowid** column to a fragmented table, the database server assigns a unique **rowid** number to each row and creates an index that it can use to find the physical location of the row. Performance using this access method is comparable to using a SERIAL, BIGSERIAL, or SERIAL8 column. The **rowid** value of a row cannot be updated, but remains stable during the existence of the row. Each row requires an additional four bytes to store the **rowid** column after you specify the WITH ROWIDS option.



Recommendation: When creating new applications, use primary keys as a method of row identification instead of using **rowid** values.

Converting a Fragmented Table to a Nonfragmented Table

You might decide that you no longer want a table to be fragmented. You can use the INIT clause to convert a fragmented table to a nonfragmented table. The following example shows the original fragmentation definition, as well as how to use the INIT clause of the ALTER FRAGMENT statement to convert the table to a nonfragmented table:

```
CREATE TABLE checks (col1 INT, col2 INT)
   FRAGMENT BY ROUND ROBIN IN dbsp1, dbsp2, dbsp3;

ALTER FRAGMENT ON TABLE checks INIT IN dbsp1;
```

You must use the IN *dbspace* clause to place the table in a dbspace explicitly.

When you use the INIT clause to change a fragmented table to a nonfragmented table, all attached indexes become nonfragmented indexes. In addition, constraints that do not use existing user-defined indexes (detached indexes) become nonfragmented indexes. All newly nonfragmented indexes exist in the same dbspace as the new nonfragmented table.

Using the INIT clause to change a fragmented table to a nonfragmented table has no effect on the fragmentation strategy of detached indexes, nor of constraints that use detached indexes.

FRAGMENT BY Clause for Tables

Use the FRAGMENT BY option of the INIT clause of the ALTER FRAGMENT statement to fragment an existing nonfragmented table, or to convert one table fragmentation strategy to another.

The PARTITION BY keywords are a synonym for the FRAGMENT BY keywords in this context.

(explicit id unique_82_Connect_42_fragtabs303) unique_82_Connect_42_fragtabs303(explicit id unique_82_Connect_42_fragtabs505)

unique_82_Connect_42_fragtabs505(explicit id unique_82_Connect_42_fragtabs606) unique_82_Connect_42_fragtabs606

FRAGMENT BY Clause for Tables

" { **FRAGMENT BY** | **PARTITION BY** } "

" { **ROUND ROBIN** { **IN** *dbspace* | **PARTITION***part* **IN***dbspace* } | **EXPRESSION** <Fragment List> | **RANGE**(*fragment_key*) <Interval Fragment Clause>[] | **LIST**(*fragment_key*) <List Fragment Clause> [] } "

Fragment List

" [**PARTITION***part*] { (*expr*) | [] **REMAINDER** } **IN***dbspace* "

Element	Description	Restrictions	Syntax
column	Column to which the strategy applies	Must exist in the table	Identifier on page 1231
dbspace	Dbspace that contains the table fragment	Must specify at least 2 but no more than 2,048 dbspaces	Identifier on page 1231
expr	Expression that defines a table fragment	Must evaluate to a Boolean value (`t` or `f`)	Expression on page 1003
part	Name of a fragment	Required for any named fragment in the same dbspace as another named fragment of the same table. Name must be unique among fragments of the same table.	Identifier on page 1231

This supports the same syntax as the FRAGMENT BY (or PARTITION BY) clause of the CREATE TABLE statement. For more information on the fragmentation strategies available for tables, see the FRAGMENT BY clause on page 356 of the CREATE TABLE statement.

Examples of range interval fragmentation

These examples define statement define range interval fragmentation strategies for an existing table. The following ALTER FRAGMENT statement defines three fragments of a range interval fragmentation strategy that includes no NULL fragment, where numeric column **c1** is the fragmentation key:

```
ALTER FRAGMENT ON TABLE T1 INIT
   FRAGMENT BY RANGE(c1)
   INTERVAL (100+100) STORE IN (dbs3, dbs4, dbs5, dbs6, dbs7, dbs8)
      PARTITION part0 VALUES < 0    IN dbs0,
```

```
      PARTITION part1 VALUES < 1000 IN dbs1,
      PARTITION part2 VALUES < 2000 IN dbs2;
```

Here the interval value expression of `(100+100` defines the interval fragment size as 200 within the range of column **c1**. No fragment is defined for fragmentation key values of 2000 or greater. If this were still the storage distribution when a row with **c1** equal to or greater than 2000 is inserted, the database server automatically creates a new fragment to store that row, which was outside the range of any existing fragment. Interval partitions are stored in the dbspaces dbs2, dbs3, dbs4, dbs5, dbs6, dbs7, and dbs8 in round-robin fashion.

The following statement similarly defines three fragments of a range interval fragmentation strategy that includes no NULL fragment, and where the DATE or DATETIME column **c2** is the fragmentation key:

```
ALTER FRAGMENT ON TABLE T1 INIT
   FRAGMENT BY RANGE(c2)
      INTERVAL (NUMTOYMINTERVAL(1,'MONTH'))
         PARTITION part0 VALUES <  DATE('01/01/2009') IN dbs0,
         PARTITION part1 VALUES <  DATE('07/01/2009') IN dbs1,
         PARTITION part2 VALUES <  DATE('01/01/2010') IN dbs2;
```

Here the interval value expression of `NUMTOYMINTERVAL(1,'MONTH')` defines the interval fragment size as a single month within the range of column **c2**. The PARTITION list defines three fragments: part0 for December of 2008, part1 for June of 2008, and part2 for December of 2009. If rows are inserted whose **c2** value is not in one of these months, the database server will create new fragments for those rows. Because no STORE IN clause is specified, the database server will store these new range interval fragments in a round-robin fashion in the dbspaces dbs0, dbs1, and dbs2 that this example lists after the IN keywords of the three PARTITION specifications.

Changing an Existing Fragmentation Strategy on a Table

You can redefine a fragmentation strategy on a table if you decide that your initial strategy does not fulfill your needs. When you alter a fragmentation strategy, the database server discards the existing fragmentation strategy and moves records to fragments as defined in the new fragmentation strategy.

The following example shows the statement that originally defined the fragmentation strategy on the table **account** and then shows an ALTER FRAGMENT statement that redefines the fragmentation strategy:

```
CREATE TABLE account (col1 INT, col2 INT)
   FRAGMENT BY ROUND ROBIN IN dbsp1, dbsp2;
ALTER FRAGMENT ON TABLE account
   INIT FRAGMENT BY EXPRESSION
   col1 < 0 IN dbsp1,
   col2 >= 0 IN dbsp2;
```

If an existing dbspace is full when you redefine a fragmentation strategy, you must not use it in the new fragmentation strategy.

Defining a Fragmentation Strategy on a Nonfragmented Table

The INIT clause can define a fragmentation strategy on a nonfragmented table, regardless of whether the table was created with a storage option.

```
CREATE TABLE balances (col1 INT, col2 INT) IN dbsp1;
ALTER FRAGMENT ON TABLE balances INIT
   FRAGMENT BY EXPRESSION col1 <= 500 IN dbsp1,
      col1 > 500 AND col1 <=1000 IN dbsp2, REMAINDER IN dbsp3;
```

When you use the INIT clause to fragment an existing nonfragmented table, all indexes on the table become fragmented in the same way as the table.

FRAGMENT BY clause for indexes

Use the FRAGMENT BY clause to redefine the storage distribution strategy of an index without redefining the index. The keywords FRAGMENT BY and PARTITION BY are synonyms in this context.

FRAGMENT BY Clause for Indexes

(explicit id unique_83_Connect_42_fragidx505) unique_83_Connect_42_fragidx505 (explicit id unique_83_Connect_42_fragidx606)

unique_83_Connect_42_fragidx606

" { **FRAGMENT** | **PARTITION** } "

" { **BY RANGE** (*fragment_key*) <Interval Fragment Clause>[] | **BY LIST** (*fragment_key*) <List Fragment Clause>[] | **BY EXPRESSION** <Expression Fragment Clause> } "

Expression Fragment Clause

" [**PARTITION***part*] (*expr*) **IN***dbspace* "

" [, [**PARTITION***part*] { **REMAINDER** | (*expr*) } **IN***dbspace*] "

Element	Description	Restrictions	Syntax
dbspace	Dbspace that contains the fragmented information	Must specify at least one (but no more than 2,048) dbspaces of the same page size	Identifier on page 1231
expr	Expression defining an index fragment	Must be unique among fragmentation expressions for the same index, and must return a Boolean value	Condition on page 967; Expression on page 1003
fragment_key	Constant expression, based on a column value. The index is fragmented on this expression.	Any column must be in the current table.	Expression on page 1003
part	Name that you declare here for the fragment. Default is the *dbspace* name.	Required for fragments in the same dbspace as another fragment of the	Identifier on page 1231

Element	Description	Restrictions	Syntax
		same index. Must be unique among fragments of the same index.	

The INIT FRAGMENT BY clause for indexes of the ALTER FRAGMENT statement can accomplish any of the following operations on the storage distribution scheme of an existing index, without redefining the index:

- Change an existing fragmented index to a nonfragmented index.
- Change the distribution scheme of an existing fragmented index to another distribution scheme of the same expression, list, or range interval type, or to a different type of distribution scheme.
- Change the interval value or the interval fragment key (or both) of a range interval distribution scheme for an existing index.

To change the interval value expression or the fragment key expression for an existing index that is fragmented by a range interval strategy, you must use the INIT FRAGMENT BY RANGE option (rather than the MODIFY clause) of the ALTER FRAGMENT statement. When you change either or both of those expressions, the Interval Fragment clause in the ALTER FRAGMENT ON INDEX statement must also define at least one range fragment.

When you use the FRAGMENT BY or PARTITION BY clause to convert an existing storage fragmentation strategy to another fragmentation strategy, Informix® discards the existing fragmentation strategy and moves the data records to fragments that you define in the new fragmentation strategy. Data movement similarly occurs when you convert a nonfragmented index to a fragmented index, and when you convert a fragmented index to a nonfragmented index.

To convert an existing fragmented index to a nonfragmented index, you can use the INIT clause to specify IN *dbspace* (or else PARTITION *partition* IN *dbspace*) as the only storage specification for a previously fragmented index.

Just as for an expression-based index fragmentation scheme that the CREATE INDEX statement defines, restrictions apply to each expression `(`*expr*`)` that you specify in the ALTER FRAGMENT ON INDEX . . . INIT FRAGMENT BY EXPRESSION statement, including these:

- Any column that the expression references must be from the current table.
- Those columns must be the indexed columns, or a subset of the indexed columns.
- The expression cannot reference fields of a column of type ROW.
- Any data values in the expression must be from only a single row.
- No subqueries, aggregates, nor **CURRVAL** or **NEXTVAL** sequence object expressions are allowed.
- The built-in **CURRENT**, **DATE**, **DBINFO**, **DBSERVERNAME**, **ROWID**, **SITENAME**, **SYSDATE**, **TODAY**, **CURRENT_USER**, and **USER** expressions are not valid in the expression.

The restrictions above also apply to fragment key expressions for list and for range interval index fragmentation schemes, including fragmentation strategies that are defined in the FRAGMENT BY clause of the CREATE INDEX statement.

Detaching an Index from a Table-Fragmentation Strategy

You can detach an index from a table-fragmentation strategy with the INIT clause of the ALTER FRAGMENT ON INDEX statement, so that an attached index becomes a detached index. This breaks any dependency of the index on the table fragmentation strategy. If the INIT clause specifies only IN *dbspace* or PARTITION *fragment* IN *dbspace* for a previously fragmented index, or if it specifies an index fragmentation strategy that differs from the storage option of the table, then the index becomes a detached index.

Fragmenting Unique and System Indexes

You can fragment unique indexes on a table that uses a round-robin or expression-based distribution scheme, but any columns referenced in the fragment expression must be indexed columns. If your index fragmentation strategy violates this restriction, the ALTER FRAGMENT INIT statement fails, and work is rolled back.

You might have an attached unique index on a table fragmented by **Column A**. If you attempt to use ALTER FRAGMENT INIT to change the table fragmentation to **Column B**, the statement fails because the unique index is defined on **Column A**. To resolve this issue, use the INIT clause on the index to detach it from the table fragmentation strategy and fragment it separately.

System indexes (such as those used in referential constraints and unique constraints) use user-defined indexes if the indexes exist. If no user-defined indexes can be used, system indexes remain nonfragmented and are moved to the dbspace where the database was created. To fragment a system index, create the fragmented index on the constraint columns and then use the ALTER TABLE statement to add the constraint.

ADD Clause

Use the ADD clause to add another fragment to a list of fragments of an existing table or index.

(explicit id unique_48_Connect_42_addclause001) unique_48_Connect_42_addclause001(explicit id unique_48_Connect_42_addclause002) unique_48_Connect_42_addclause002(explicit id unique_48_Connect_42_addclause003) unique_48_Connect_42_addclause003(explicit id unique_48_Connect_42_addclause004) unique_48_Connect_42_addclause004

ADD Clause

" ADD "

" { [[PARTITION *part*] REMAINDER IN] [PARTITION *part* IN] *dbspace* | [PARTITION *part*] <ADD Expression> IN *dbspace* [{ BEFORE | AFTER } *fragment*] | INTERVAL [STORE] IN(*new_dbspace*) } "

ADD Expression

" { { VALUES < *range_expr* | VALUES IS NULL } | *expression* | VALUES { *const_expr* | [IS] NULL } } "

Element	Description	Restrictions	Syntax
dbspace	Name of a dbspace to store the new fragment	Must exist If the table is in a tenant database, the *dbspace* must be a dedicated dbspace in the tenant database properties list. If the table is not in a tenant database, the *dbspace* cannot be the name of a dbspace that is dedicated to a tenant database.	Identifier on page 1231
expression	Expression that defines the new fragment that is to be added	Must return a Boolean value (`t` or `f`)	Condition on page 967; Expression on page 1003
fragment	Name of an existing fragment	Must exist	Identifier on page 1231
new_dbspace	Name of a dbspace being added to store new interval fragments	Must exist If the table is in a tenant database, the *new_dbspace* must be a dedicated dbspace in the tenant database properties list. If the table is not in a tenant database, the *new_dbspace* cannot be the name of a dbspace that is dedicated to a tenant database.	Identifier on page 1231
part	Name that you declare here for the new fragment. The default is the name of the dbspace.	Required if another fragment in the fragment list is stored in the same dbspace. Must be unique among names of fragments of the index.	Identifier on page 1231

The *expression* can contain *column* names only from the current table, and data values only from a single row. No subqueries or aggregates are allowed. In addition, the built-in **CURRENT**, **DATE**, **DBINFO**, **SYSDATE**, and **TODAY** expressions are not valid in this context.

Adding a New Dbspace to a Round-Robin Distribution Scheme

You can add more dbspaces to a round-robin distribution scheme. The following example shows the original round-robin definition:

```
CREATE TABLE book (col1 INT, col2 INT)
   FRAGMENT BY ROUND ROBIN IN dbsp1, dbsp4;
```

To add another dbspace, use the ADD clause, as in this example:

```
ALTER FRAGMENT ON TABLE book ADD dbsp3;
```

Adding a New Named Fragment to a Round-Robin Distribution Scheme

In Informix®, you can add a named fragment to an existing round-robin distribution scheme. The name must be unique within the distribution among fragments of the same dbspace. The following example specifies the same original round-robin fragmentation definition as in the previous section:

```
CREATE TABLE book (col1 INT, col2 INT)
   FRAGMENT BY ROUND ROBIN IN dbsp1, dbsp4;
```

To add a new named fragment, you can use the ADD clause, as in the following example:

```
ALTER FRAGMENT ON TABLE book
   ADD PARTITION chapter3 IN dbsp1;
```

The new distribution uses **dbsp1**, **dbsp4**, and **chapter3** as storage locations for a 3-part round-robin fragmentation scheme. The records in the fragment **chapter3** are stored separately in the dbspace **dbsp1** from the records in the first fragment, which is identified only by the dbspace name: **dbsp1**. The newly-added **chapter3** fragment will be favored exclusively by inserts and loads until it no longer has the fewest rows among the table's fragments.

Adding an expression-based fragment

Adding a fragment expression to the fragmentation list of an expression-based distribution scheme can relocate records from existing fragments into the new fragment. When you insert a new fragment into the fragmentation list, the database server reevaluates all the data in the existing fragments that follow the new fragment. (The **evalpos** column value for any fragment in the **sysfragments** system catalog table indicates the ordinal position of that fragment within the fragmentation list.)

The next statement fragment shows the original expression definition:

```
FRAGMENT BY EXPRESSION
   c1 < 100 IN dbsp1, c1 >= 100 AND c1 < 200 IN dbsp2,
   REMAINDER IN dbsp3
```

To add another named fragment in dbspace **dbsp2** to hold rows for column **c1** values between `200` and `299`, use the following ALTER FRAGMENT statement:

```
ALTER FRAGMENT ON TABLE news
   ADD PARTITION century3 (c1 >= 200 AND c1 < 300) IN dbsp2;
```

Any rows that were formerly in the remainder fragment but that fit the criteria (`c1 >= 200` AND `c1 < 300`) move to the new **century3** fragment in dbspace **dbsp2**.

If the ALTER FRAGMENT ADD operation results in the redistribution of data rows while the automatic mode for updating distribution statistics is enabled, the database server drops the distribution statistics of the affected fragments, but the table

statistics are not dropped. The next query on the table will cause the database server to recalculate the statistics for the same fragments.

Using the BEFORE and AFTER Options

The BEFORE and AFTER options can position the new fragment either before or after an existing fragment within the fragmentation list. The name of a fragment is the name of the dbspace or the name declared in the PARTITION clause. You cannot use the BEFORE and AFTER options if the distribution scheme is round-robin or range interval.

When you attach a new fragment without an explicit BEFORE or AFTER option, the database server places the added fragment at the end of the fragmentation list, unless a remainder fragment exists. If a remainder fragment exists, the new fragment is placed immediately before the remainder fragment. You cannot position a new fragment after the remainder fragment.

Using the REMAINDER Option

You cannot add a remainder fragment if one already exists. If you add a new fragment when a remainder exists, the database server retrieves and reevaluates all records in the remainder fragment; some records might move to the new fragment. The remainder fragment is always the last item in the fragment list.

You cannot add a remainder fragment to the fragmentation list of a range interval fragmentation scheme.

DROP Clause

Use the DROP clause to remove an existing fragment from the fragmentation list of a table or index that is fragmented by round-robin. For a table or index that is fragmented by range interval, you can use this clause to drop one or more dbspaces from the list of dbspaces that store system-generated interval fragments.

Use the DETACH clause, rather than the DROP clause, to remove an existing fragment from a table that uses range-inerval fragmentation, such as a rolling window table.

(explicit id unique_49_Connect_42_afdrop002) unique_49_Connect_42_afdrop002

DROP Clause

" **DROP** { **[PARTITION]** *fragment* | **INTERVAL [STORE] IN(** *dbspace*) } "

Element	Description	Restrictions	Syntax
dbspace	Dbspace that stores system generated fragments	Must exist when you execute the statement.	Identifier on page 1231
fragment	Name of the fragment	Must exist when you execute the statement. For list or range interval	Identifier on page 1231

Element	Description	Restrictions	Syntax
		fragments, the PARTITION keyword must precede this name.	

If the table is fragmented by expression, you cannot drop a fragment containing data that cannot be moved to another fragment. If the distribution scheme has a REMAINDER option, or if the expressions overlap, you can drop a fragment that contains data. You cannot, however, drop a fragment if the table has only two fragments.

When you want to make a fragmented table nonfragmented, you can use either the INIT clause or the DETACH clause of the ALTER FRAGMENT statement, rather than the DROP clause.

If the *fragment* was not given a name by the user who created the table or index or added the fragment, then the name of the dbspace is also the name of the fragment. If the fragment is a system-generated range interval fragment of a table or of an index, its name is **sys_p***evalpos*, where *evalpos* is the **sysfragments.evalpos** entry for the fragment in the system catalog. If a table and its index use the same range interval fragmentation strategy, each system-generated index fragment has the same identifier as a system-generated fragment of the table.

When you drop a fragment, the database server attempts to move all records in the dropped fragment to another fragment. In this case, the destination fragment might not have enough space for the additional records. If this happens, follow one of the procedures that ALTER FRAGMENT and Transaction Logging on page 25 describes to increase your available space, and retry the ALTER FRAGMENT operation.

When the DROP clause specifies one or more dbspaces to remove from a range interval fragmentation strategy, those dbspaces are not affected, but the database server moves the data in any fragments of the table or index that are stored in those dbspaces to other available dbspaces. (The range interval fragmentation strategy is also affected, because it no longer includes the specified dbspaces among its storage locations for new system-generated fragments.)

You cannot use the DROP clause to drop a range interval fragment that contains data.

You can use this clause to drop a list fragment that contains data only if a remainder fragment exists to receive the data.

If the fragmented table has fragment level statistics, the ALTER FRAGMENT DROP operation also drops any fragment-level statistic distribution for the fragment been dropped. Table-level statistics, however, are not recalculated. The next explicit or automatic UPDATE STATISTICS operation on the table will rebuild stale fragment-level distributions and merge them to form table level distributions and store the results in the system catalog.

Examples of the DROP clause of the ALTER FRAGMENT statement

The following examples show how to drop a fragment from a round-robin fragmentation list. The first line shows how to drop an index fragment, and the second line shows how to drop a table fragment.

```
ALTER FRAGMENT ON INDEX cust_indx DROP dbsp2;

ALTER FRAGMENT ON TABLE customer DROP dbsp1;
```

The following examples each drop a fragment that was defined by list fragmentation strategies. The first line shows how to drop a table fragment, and the second line shows how to drop an index fragment.

```
ALTER FRAGMENT ON TABLE T2 DROP PARTITION part4;

ALTER FRAGMENT ON INDEX idx2 DROP PARTITION part4;
```

In both examples above, the PARTITION keyword is required, and the name of the dropped fragment is **part4**. If the index **idx2** is defined on table **T2** and thus the same storage distribution strategy as table **T2**, the second statement is not necessary, because the database server automatically modifies the fragmentation strategy of an attached index when the table fragmentation list is modified. If these fragments are not empty, the database server moves their data to the remainder fragments (or returns an error, if no remainder fragment exists).

The following examples each drop two dbspaces that store system-defined interval range fragments, as defined by range interval fragmentation strategies. The first statement drops dbspaces **dbs7** and **dbs8** from the storage spaces for table fragments, and the second statement drops the same storage spaces for index fragments:

```
ALTER FRAGMENT ON TABLE T1 DROP INTERVAL STORE IN (dbs7, dbs8);

ALTER FRAGMENT ON INDEX idx1 DROP INTERVAL STORE IN (dbs7, dbs8);
```

The PARTITION keyword is again required, and if **idx1** is an attached index on table **T1**, the second statement is not necessary: When the table fragmentation list is modified, the database server automatically modifies the fragmentation strategy of any attached index to match the modified strategy of its table. If these fragments are not empty, the database server moves any fragments from the specified **dbs7** and **dbs8** dbspaces to other available dbspaces.

MODIFY Clause

Use the MODIFY clause to change the fragmentation list or the fragment key of a table or of an index that is fragmented by list or by expression, or to define, modify, or disable a range-interval or rolling window fragmentation scheme.

Use the MODIFY clause to change the existing fragmentation list of a table or of an index. You can use this clause to accomplish one or more of the following tasks:

- Move an existing fragment from one dbspace to a different dbspace
- Change the expression associated with an existing list-based or expression-based fragment
- Change the expression that defines the transition fragment in a range-interval fragmentation list
- Rename one or more existing fragments
- Enable or disable the automatic creation of interval fragments
- Replace the list of dbspaces or the function specifying where to store new interval fragments
- Change a rolling window table to a table fragmented by interval (with no purge policy)
- Change a table that is fragmented by range interval to a rolling window table
- Change the purge policy of a rolling window table by one or more of these actions:
    - Resetting the limit on the number of interval fragments,
    - Changing the allocated storage size limit for the table,
    - Replacing the `ATTACH` or `DISCARD` keyword option,
    - Replacing the `ANY` or `INTERVAL FIRST` or `INTERVAL ONLY` keyword option.

The MODIFY clause of the ALTER FRAGMENT statement has the following syntax:

(explicit id unique_50_Connect_42_mod001) unique_50_Connect_42_mod001(explicit id unique_50_Connect_42_mod002)

unique_50_Connect_42_mod002(explicit id unique_50_Connect_42_mod003) unique_50_Connect_42_mod003(explicit id unique_50_Connect_42_mod004)

unique_50_Connect_42_mod004

MODIFY Clause

(explicit id unique_50_Connect_42_mod003c) unique_50_Connect_42_mod003c

```
" MODIFY { [PARTITION] old TO PARTITION new <Fragment Expression> [ IN dbspace ] | [ INTERVAL { [ { ENABLED | DISABLED } ] [STORE ] IN( { dbspace | dbspace_fun( ) } ) | { ENABLED | DISABLED } | TRANSITION TO const_expr | <Rolling Window clause> ] } } "
```

Fragment Expression

```
" { expression | [ { VALUES < range_expr | [ VALUES IS NULL } | [ VALUES { const_expr | [ [IS] NULL } | [ [ REMAINDER } "
```

Element	Description	Restrictions	Syntax
const_expr	Constant expression that defines the list of values for a fragment to store, or a new upper limit for the range-interval transition fragment	Must be a quoted string or a literal value. For fragmentation by list, each value must be unique among the expression lists for fragments of the same object.	Constant Expressions on page 1040
dbspace	Dbspace that stores the *new* fragment	Must exist at time of execution. All of the dbspaces must have the same page size. If the table is in a tenant database, the *dbspace* must be a dedicated dbspace in the tenant database properties list. If the table is not in a tenant database, the *dbspace* cannot be the name of a dbspace that is dedicated to a tenant database.	Identifier on page 1231
dbspace_fun	Name of a UDF that returns the name of a dbspace	The user-defined function and the returned dbspace must exist when the database server calls the UDR to allocate storage for a new fragment.	CREATE FUNCTION statement on page 225
expression	Modified expression	Can specify columns in current table only and data from only a single row	Condition on page 967;

Element	Description	Restrictions	Syntax
			Expression on page 1003
new_dbspace	Dbspace that stores system-generated range interval fragments	Must exist at time of execution. All of the dbspaces must have the same page size. If the table is in a tenant database, the *new_dbspace* must be a dedicated dbspace in the tenant database properties list. If the table is not in a tenant database, the *new_dbspace* cannot be the name of a dbspace that is dedicated to a tenant database.	Identifier on page 1231
new	Name that you declare here for the modified fragment	Must be unique among fragment names in the fragmentation list. If a table and its index use the same range interval or list fragmentation strategy, each index fragment must have the same name as the corresponding table fragment.	Identifier on page 1231
old	Name of an existing fragment	Must exist in the fragmentation list. For list or range interval fragments, the PARTITION keyword must precede this name.	Identifier on page 1231
range _expr	Range expression. This constant expression defines the upper bound for fragment key values stored in the fragment	Must be a constant literal expression that evaluates to a numeric, DATETIME, or DATE data type compatible with the data type of the fragment key expression. See also Restrictions on the MODIFY clause for range interval fragments on page 56.	Constant Expressions on page 1040

Usage

Here *dbspace* and *old* (or *old* and *new*) can be identical, if you are not changing the storage location. For tables or indexes that are fragmented by range interval, the specified list of dbspaces that follows the STORE IN keywords replaces the list of dbspaces that was in effect before you issued the ALTER FRAGMENT . . . MODIFY statement. Fragments in the former list of dbspaces are not relocated by this option.

Instead of a list of literal dbspace identifiers, the STORE IN clause can optionally specify a user-defined function that returns the name of an existing dbspace. The identifier that you declare for this UDF is arbitrary. For more information about this UDF and an example of how to create it, see the discussion of the STORE IN clause in the CREATE TABLE topic #unique_87.

To use the MODIFY clause both to change the *expression* and to move its corresponding fragment to a new storage location, you must change the *expression* and you must also specify the name of a different *dbspace* or partition.

You must declare a *new* fragment name if more than one fragment of the same table or index has the same identifier as the *dbspace*. The PARTITION keyword is required immediately before the *new* fragment name for range interval fragments and for list fragments, but it is optional for round-robin fragments and expression-based fragments.

The *expression* must evaluate to a Boolean value (true or false).

No subqueries or aggregates are allowed in the *expression*. In addition, the built-in **CURRENT**, **DATE**, **DBINFO**, **SYSDATE**, and **TODAY** expressions are not valid.

When you use the MODIFY clause to change an expression without changing the storage location for the expression, you must use the same name for the *old* and the *new* fragment. If the dbspace consists of only a single partition, however, you can specify the same name for *old* and for *dbspace*, as in the following example:

```
ALTER FRAGMENT ON TABLE cust_acct
   MODIFY dbsp1 TO acct_num < 65 IN dbsp1;
```

For list fragmentation strategies, the ALTER FRAGMENT MODIFY statement fails with an error if a new list expression overlaps any existing list expressions for another fragment of the same table or index.

When you use the MODIFY clause to move a fragment from one dbspace to another, *old* is the name of the dbspace where the fragment was previously located, and *dbspace* is the new location for the fragment, as in the following example:

```
ALTER FRAGMENT ON TABLE cust_acct
   MODIFY PARTITION part1 TO PARTITION part2 (acct_num < 35) IN dbsp2;
```

The ALTER FRAGMENT statement above modifies the distribution scheme for the **cust_acct** table, so that the rows with values in column **acct_num** that are less than 35 (and that had previously been assigned to fragment **part1** that was stored in the dbspace **dbsp1**) will now be assigned to the fragment **part2** that is stored in the dbspace **dbsp2**.

When you use the MODIFY clause, the underlying dbspaces are not affected. Only the data values within the fragments or dbspaces are affected.

Unless the fragmentation strategy is by range interval, if no remainder fragment already exists, you can redefine a nonremainder fragment as a remainder fragment for rows that do not match the fragment key values for any other fragment. You cannot change a REMAINDER fragment into a nonremainder fragment, however, if any rows already stored in the REMAINDER fragment do not satisfy the new *expression*.

An *attached* index has the same storage distribution as its table. If all the indexes on a table are attached indexes, and you use the MODIFY clause to modify the table fragments, the database server automatically modifies the storage distribution strategy for the index to match the new table fragmentation strategy.

The *old* specification cannot reference the transition fragment (the last range fragment) of any table that is fragmented by a range-interval storage distribution scheme. The only modification that is valid for that fragment is to use the TRANSITION TO *const_expr* clause to increase the transition value. For any other syntax that attempts to redefine the range expression of

the transition fragment directly, the database server returns an error. For more information, see the topic Using the MODIFY INTERVAL TRANSITION option on page 63.

Restrictions on the MODIFY clause for range interval fragments

The MODIFY clause of the ALTER FRAGMENT statement cannot change the interval value, nor the fragment key. To change either of these elements of the range interval storage distribution scheme, you must use the INIT option of the ALTER FRAGMENT statement.

The MODIFY clause cannot change the value of the range expression of a fragment if any of the following are true:

- The fragment is the last fragment, and the new value is smaller than the current value.
- The new value overlaps the boundary of an existing fragment.
- The fragment is a system-generated interval fragment.

You can modify the value of a user-defined range fragment, but the new value cannot cross adjacent fragment boundaries, and the database server must be able to accomplish any data movement that the new range expression implies.

The MODIFY clause can change the list of storage spaces where an existing fragment is stored, and it can also change the list of storage spaces where new system-generated interval fragments will be stored, but the same MODIFY clause cannot accomplish both tasks. To change both lists, you must issue two separate ALTER FRAGMENT . . . MODIFY statements.

Similarly, a MODIFY clause that enables or disables the current range interval distribution scheme cannot also move an existing range interval fragment to a different dbspace, or create a new user-defined fragment. Separate ALTER FRAGMENT . . . MODIFY statements are required for each of these tasks.

For range fragments of tables and indexes fragmented by interval, you can modify the fragment expression for the first and for intermediate range fragments. Any overlaps are resolved by moving data, so that fragment key values of the rows stored in the redefined range fragments are non-overlapping. For the last range fragment, however, you can modify the transition value in its fragment expression only if the new range expression satisfies all of the following conditions:

- It does not partially or completely match any existing interval fragment expression.
- It will not partially match any future interval fragment expressions that the system can generate automatically.
- Any gap that the new transition value leaves between fragments must be an integer multiple of the *intvl_expr* interval value.

You cannot define a remainder fragment for a table that is fragmented by range interval.

If you use the MODIFY clause to rename an existing fragment, the new name cannot begin with the characters `sys_p`.

Range, interval, and transition fragments

For objects that use a range interval storage distribution strategy, it is useful to distinguish among three types of fragments:

- A *range* fragment is a fragment whose name, fragment-key expression, and storage location are defined explicitly in the Interval Fragment clause within the table or index definition. Range interval fragmentation requires that at least one range fragment be defined.
- An *interval* fragment is a fragment whose name, fragment-key expression, and storage location are defined automatically by the database server when an insert or load operation attempts to store a row whose fragment-key value is false for the fragment-key expression of every existing fragment.
- The range fragment whose upper limit in its VALUES clause is larger than for the fragment-key expression for any other range fragment is called the *transition* fragment. The upper limit specified in the VALUES clause of the transition fragment is called the *transition value* for the table. If no interval fragments have been created for the object, inserting a row whose fragment-key value exceeds that transition value requires the database server to create a new interval fragment.

Operations that the MODIFY clause of the ALTER FRAGMENT statement can perform on transition fragments are more restricted than MODIFY operations on other range and interval fragments.

The ALTER FRAGMENT MODIFY statement cannot change the range expression that defines a transition fragment unless you also include the MODIFY TRANSITION keywords.

The database server cannot create interval fragments unless the Interval Fragment clause within the table or index definition defined a range interval fragment key, and the fragmentation scheme is not currently disabled by the ALTER FRAGMENT . . . MODIFY INTERVAL DISABLE statement.

Restrictions on modifying rolling window tables

The Rolling Window clause of the ALTER FRAGMENT MODIFY INTERVAL statement cannot define a purging policy on a table that has any of the following attributes:

- The table has a ROWID shadow column.
- Another table has a foreign key constraint that references a PRIMARY KEY in the table.

Rolling Window clause

Use the Rolling Window clause of the ALTER FRAGMENT MODIFY INTERVAL statement to add a purge policy to a table that uses range-interval distributed storage, or to modify or drop the purge policy of a rolling window table.

For tables with range-interval distributed storage, the Rolling Window clause supports the following syntax to drop all purge policy limits on interval fragments, or to set new purge policy limits on interval fragments:

Rolling Window clause of ALTER FRAGMENT

```
ALTER FRAGMENT FOR TABLE table MODIFY INTERVAL <Rolling Window clause>
```

Case 1: Destroy all Purge Policies for a table

```
" DROP ALL ROLLING "
```

Case 2: Limit the maximum allocated storage size

```
" [ ROLLING( quantity FRAGMENTS) ] LIMIT TO size units "
```

```
" { { DETACH | DISCARD } { INTERVAL FIRST | [ { ANY | INTERVAL ONLY } ] } } "
```

Case 3: Limit only the quantity of interval fragments

```
" ROLLING( quantity FRAGMENTS) { DETACH | DISCARD } "
```

Element	Description	Restrictions	Syntax
quantity	Upper limit in purge policy on the number of interval fragments	Must be an integer greater than zero. User-defined range fragments are not included in this limit.	Literal integer
size	Upper limit on the total allocated storage for the table and its indexes	Must be greater than zero	Literal integer
table	Name of an existing table with range-interval fragmentation	Must not have a ROWID column or a primary-key constraint that a foreign-key constraint references. Any index must have the same storage distribution.	Identifier on page 1231
units	Abbreviated unit of total mass storage for the table. Any trailing characters cause a syntax error.	Must be `K`, `KB`, `KiB`, `M`, `MB`, `MiB`, `G`, `GB`, `GiB`, `T`, `TB`, `TiB` (case insensitive).	Unquoted character string, beginning with the letter `K`, `M`, `G`, or `T`

Usage

The syntax of the Rolling Window clause in ALTER FRAGMENT MODIFY INTERVAL statement supports a superset of the syntax of the Rolling Window clause in CREATE TABLE FRAGMENT BY INTERVAL statements.

Modifying rolling window tables

The Rolling Window clause of the ALTER FRAGMENT MODIFY INTERVAL statement resembles in its syntax, but is not identical to, the Rolling Window clause of the CREATE TABLE statement. The Rolling Window clause of ALTER FRAGMENT supports the following functionality:

- You can define a purge policy for a table that uses range-interval fragmentation.
- You can modify an existing purge policy by any of the following changes:
  - Changing the `ROLLING FRAGMENTS` value for *quantity*
  - Changing the `LIMIT TO` value for *size*
  - Replacing the `DETACH` or `DISCARD` keyword with the `DETACH` or `DISCARD` keyword.
  - Replacing the `ANY` or `INTERVAL FIRST` or `INTERVAL ONLY` keyword option.

  If you delete the `ANY` or `INTERVAL FIRST` or `INTERVAL ONLY` keyword specification without replacement, the default purge policy action is `INTERVAL FIRST`. (For more information about the effects of these keywords on which qualifying fragments will be purged, see the topic #unique_87.)
- You can specify the `INTERVAL DISABLED` keywords to disable interval fragmentation for a rolling window table, thereby suspending its purge policy.
- You can specify the `INTERVAL ENABLED` keywords to restore interval fragmentation (and re-enable the purge policy) for a table for which interval fragmentation and the creation and archiving or destruction of rolling fragments had been disabled.
- You can specify the `DROP ALL ROLLING` keywords to remove an existing purge policy. The effect is to change the rolling window table to a table fragmented by interval.

If you intend to suspend the current purge policy temporarily, and subsequently to restore the same purge policy, you should use the `INTERVAL DISABLED` keywords, rather than the `DROP ALL ROLLING` keywords.

Enforcing a purge policy

A rolling window table's purge policy is not immediately enforced when the total allocated storage size or the total number of interval fragments exceeds the limit that the Rolling Window clause specifies.

Purge policies are designed to be enforced daily as a Scheduler task at a time when the required DETACH and ATTACH operations on fragments of the rolling window table are unlikely to conflict with access attempts by concurrent users. By default, purge policies are enforced daily, at 00:45 local time. For more information, see the description of the built-in purge_tables task of the Scheduler in your *HCL® Informix® Administrator's Guide*.

Purge policies also can be manually enforced by running the syspurge() system function. After the DBA invokes the syspurge() function, the database server inspects the system catalog, and identifies any rolling window tables whose purging policy has been exceeded. The database server then either discards or detaches, as specified by the purge policy, qualifying rolling fragments until the purging policy is satisfied, or until no more rolling fragments can be removed. The syspurge() function requires no arguments, but accepts an optional argument that enables online log diagnostics.

Only users with DBA access privileges can call routines that implement the `DETACH` or `DISCARD` options for detached rolling fragments. Users with RESOURCE access privileges can execute the syspurge() function, but this can only enforce purging policies on tables that they own.

The database server silently ignores any invocation of the syspurge() function on secondary servers in High Availability Data Replication (HDR) cluster environments. Similarly, in grid environments, purge policies on replicated tables are not enforced. This is because grid environments and cluster environments do not replicate ALTER FRAGMENT changes that the `DETACH` and `DISCARD` options trigger, which are at the core of rolling window purge policies.

The Rolling Window clause provides two keyword options for processing detached rolling interval fragments:

- Use `DETACH` to attach the fragments into independent tables that the database server automatically creates, and whose table identifiers are of this form:

  ```
  < original_table_name >_< lower value >_< higher value >
  ```

  Here *lower_value* and *higher_value* are the minimum and maximum values of the interval range for that fragment, before it was detached.

  If a table of that name already exists, a numeric counter is appended after the higher value, beginning with `_1` for the first additional table:

  ```
  < original_table_name >_< lower value >_< higher value >_1
  ```

  and so forth, with `_2` appended to the next table name (or a larger integer is appended, if appending `_2` does not produce a unique table name).
- Use `DISCARD` to destroy the detached fragments.

  The `DISCARD` keyword specifies that successfully detached fragments be dropped, so that when the purge policy is enforced, unneeded data records are removed in a timely manner. In this way, the number of rolling fragments or the total amount of storage space for the rolling window table is constrained to the stipulated value

Examples of modifying an existing rolling window table

The examples of ALTER FRAGMENT ON TABLE . . . MODIFY INTERVAL statements that follow all change the current rolling-window options of a table called **window_orders** that the following CREATE TABLE statement defined:

```
CREATE TABLE window_orders
       (order_id INT, cust_id INT,
        order_date DATE, order_desc CHAR (1024))
   FRAGMENT BY RANGE (order_date)
   INTERVAL (NUMTOYMINTERVAL (1,'MONTH'))
      ROLLING (4 FRAGMENTS) DETACH
      STORE IN (dbs1, dbs2, dbs3)
      PARTITION p0 VALUES < DATE ('01/01/2015') IN dbs1,
      PARTITION p4 VALUES IS NULL in dbs3;
```

Here the **window_orders** table uses a range-interval distributed storage strategy

- a range fragment **p0** for fragment-key values earlier than the year 2015,
- and three rolling-window interval fragments,
- and fragment **p4** as a NULL fragment to store rows with no value in the **order_date** fragment-key column.

Because the interval within the range of this fragment key is defined as one month, and the interval transition value is the first day of the year 2015, the first interval fragment will be generated when a record is inserted with an **order_date** value in a year later than 2014. Successive interval fragments will be stored in the dbspaces **dbs1**, **dbs2**, and **dbs3** in round-robin fashion.

In the CREATE TABLE statement example above, the Rolling Window clause sets at `3` the maximum number of rolling interval fragments. If rows are be added in each of the first three months of 2015, three rolling fragments will be generated by March of that year, because each new interval fragment stores data from only a single month. If a 5th interval fragment is created in May, this will exceed the purge policy limit on rolling fragments. Because no limit on storage size is specified, the default `INTERVAL FIRST` criterion will detach the interval fragment whose **evalpos** value is smallest among the four rolling fragments. That fragment will be attached to another table, rather than destroyed, because the purge policy specifies `DETACH`, rather than `DISCARD`.

ALTER FRAGMENT ON TABLE **window_orders** MODIFY INTERVAL statements that follow illustrate various changes to the original Rolling Window distribution scheme, as defined above for the **window_orders** table.

- Set a maximum size limit on the data stored in the table:

  ```
  ALTER FRAGMENT ON TABLE window_orders MODIFY INTERVAL
     LIMIT TO 30 MiB DETACH INTERVAL ONLY;
  ```

  This sets a 30 megabyte limit on the total storage size of the **window_orders** table, and stipulates that range fragments like **p0** cannot be detached to reduce the current size below that limit. The maximum number of interval fragments (`4`) and the disposal mode of the purge policy (`DETACH`) are unchanged. Note that the actual storage size could exceed the new limit until the Scheduler enforces the purge policy. At that point, however, at least one interval fragment would be detached, in an effort to comply with the `LIMIT TO` setting.

  The `ANY`, or `INTERVAL FIRST`, or `INTERVAL ONLY` option for the fragments to purge cannot be reset unless you also include a `LIMIT TO` specification in the Rolling Window clause of the ALTER FRAGMENT MODIFY statement.

- Changing the disposal option:

  ```
  ALTER FRAGMENT ON TABLE window_orders MODIFY INTERVAL
     ROLLING (4 FRAGMENTS) DISCARD;
  ```

  This maintains the current number of interval fragments, but changes the disposal option to `DISCARD`, so that the detached will be dropped from the database, in accordance with this change to the original purge policy for the **window_orders** table.

- Changing the number of interval fragments:

  ```
  ALTER FRAGMENT ON TABLE window_orders MODIFY INTERVAL
     ROLLING (6 FRAGMENTS);
  ```

  This increases the number of interval fragments to six, so that no new fragment is added (and no existing fragment is detached) until a new row is inserted whose fragment-key value is a DATE whose month is seven months later than the month values in he oldest fragment.

- Rolling back both previous changes to the quantity and disposal of interval fragments:

```
ALTER FRAGMENT ON TABLE window_orders MODIFY INTERVAL
   ROLLING (4 FRAGMENTS) DETACH;
```

  This restores the original **window_orders** storage options, and illustrates that a single ALTER FRAGMENT ON TABLE . . . MODIFY INTERVAL statement can change more than one option of a Rolling Window strategy.
- Drop the rolling-window distributed-storage behavior of the table:

```
ALTER FRAGMENT ON TABLE window_orders MODIFY INTERVAL
   DROP ALL ROLLING;
```

  This drops the automatic purge policy, changing **window_orders** from a rolling-window table to an ordinary range-interval table. If a `LIMIT TO` maximum size, or if `ANY` or other keyword option priorities for dropping fragments had been part of the purge policy, those would have also been dropped. No existing data is destroyed, but the future distributed-storage behavior of the **window_orders** table will match what the following CREATE TABLE statement (with no Rolling Window clause) implies:

```
CREATE TABLE window_orders
       (order_id INT, cust_id INT,
        order_date DATE, order_desc CHAR (1024))
   FRAGMENT BY RANGE (order_date)
   INTERVAL (NUMTOYMINTERVAL (1,'MONTH'))
      STORE IN (dbs1, dbs2, dbs3)
      PARTITION p0 VALUES < DATE ('01/01/2015') IN dbs1,
      PARTITION p4 VALUES IS NULL in dbs3;
```

Restrictions on rolling window tables

The ALTER FRAGMENT MODIFY statement cannot use the Rolling Window clause to change a table that has any of the following attributes into a rolling window table:

- a ROWID column
- a column or columns defined as the primary key of a referential constraint.
- a detached index (that is, an index whose storage distribution scheme is not identical to the fragmentation strategy of the table).

Similarly, the ALTER TABLE statement cannot add a ROWID column or a primary-key constraint to a rolling window table.

- The purging strategy that the Rolling Window clause defines for rolling fragments requires the database server to perform ALTER FRAGMENT DETACH operations on fragments that satisfy the `DETACH` or `DISCARD` criteria. The ALTER FRAGMENT DETACH statement is disallowed, however, on tables with primary keys that are referenced by an enabled foreign key constraint, or on tables with ROWIDs. For this reason, the CREATE TABLE and ALTER FRAGMENT ON TABLE . . . MODIFY INTERVAL statements cannot define or modify a purging policy on tables that have primary key constraints or ROWID shadow columns.
- Any index defined on a rolling window table must have the same range-interval storage distribution as the rolling window table.

Using the MODIFY INTERVAL TRANSITION option

You can use this option to increase the transition value of the last range fragment of a table that has a range-interval fragmentation scheme. The transition value cannot be decreased by using this MODIFY INTERVAL TRANSITION option to the ALTER FRAGMENT statement.

You cannot use the PARTITION *partition* VALUES syntax of the MODIFY option to modify the range expression for the last range fragment (also called the *transition fragment*) of a table that uses a range-interval storage distribution scheme. The transition value, however, which is the upper limit in the range expression, can be increased by using the MODIFY INTERVAL TRANSITION TO keywords to specify the new upper limit. There is no data movement when the transition value is changed.

To decrease the transition value (by resetting the upper limit on the range of the transition fragment), you must perform an `ALTER FRAGMENT INIT` operation to redefine the range-interval distributed storage scheme for the table.

Automatic fragment renaming when the transition value increases

ALTER FRAGMENT statements that specify the MODIFY INTERVAL TRANSITION option can cause existing fragments to be renamed:

- If there are no interval fragments between the new and old transition values, but interval fragments already exist above the new transition value, the digit that terminates the system-generated interval fragment name is reduced by the number of interval fragment boundaries occupied by the difference between the new and old transition values.

  For example, if the interval value expression defining an interval size evaluates to 20, and the difference between the old transition value and the new transition value is 60, then an interval fragment whose name is **sys_p7** is renamed to **sys_p4**, because the quotient is `(60/20) = 3`.

- If interval fragments exist between the new and old transition values, the characters **rg** are appended to their names to indicate that they have become range fragments, because the upper limit of their fragment expression is no longer greater than the transition value for the table.

  For example, if the transition value of a table were increased to match the upper VALUES limit of its interval fragment **sys_p5**, that fragment would be changed to a range fragment, and renamed **sys_p5rg**. (It would also become the transition fragment.) If another interval fragment called **sys_p4** also had a smaller VALUES upper limit in its fragment expression, that fragment would also become a range fragment, and would be renamed **sys_p4rg**.

During a fragment renaming operation, an exclusive lock is placed on the fragment while the **sysfragments** system catalog table is being updated

- with new fragment-identifier values in the **partition** column,
- and with new integer values in the **evalpos** column for any interval fragments or rolling interval fragments whose ordinal positions in the fragment list changed during the current ALTER FRAGMENT MODIFY operation.

In the cases listed above, some fragments are renamed to ensure that every fragment name in the fragment list is unique, and to maintain the correlation between system-generated names for interval fragments and the corresponding

sysfragments.evalpos value for those fragments in the system catalog. (See also the section Automatic renaming of interval fragment identifiers on page 26.)

Several of the ALTER FRAGMENT examples that follow illustrate this fragment-renaming behavior.

Example of ALTER FRAGMENT MODIFY INTERVAL TRANSITION

The following statements define a fragmented **tabtrans** table that uses a range-interval storage distribution scheme, with the integer column **i** the fragment key, and an interval value of `100`. The transition fragment **p2** has a transition value of `300`, meaning that the database server will define a new interval fragment during any operation on the table to store a new row with a fragment key value of `300` or greater.

```
CREATE TABLE tabtrans (i INT, c CHAR(2))
   FRAGMENT BY RANGE (i)
   INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
      PARTITION p0 VALUES < 100 IN dbs0,
      PARTITION p1 VALUES < 200 IN dbs1,
      PARTITION p2 VALUES < 300 IN dbs0;  -- last range fragment (also
                                          -- called transition fragment)
```

Examples that follow are based on this **tabtrans** table.

The following ALTER FRAGMENT statement attempts to increase the transition fragment value from `300` to `250`:

```
ALTER FRAGMENT ON TABLE tabtrans
   MODIFY INTERVAL TRANSITION TO 250;
```

This statement fails, because it attempts to reduce the transition value. If the design goal is to keep the current interval value of `100`, but for the new transition value to become `250`, an ALTER FRAGMENT INIT operation is required to redefine the range fragments. To keep the range-fragment boundaries aligned, the new upper limit for the range fragment immediately preceding the transition fragment must be `150`. In the new distributes storage scheme, if a row with a fragment-key value greater than `250` is inserted into the table, the database server generates a new interval fragment with a range of `100`, and with an integer value of the form 50 (modulo 100) as the upper limit.

If there are no interval fragments between the new and old transition values, the database server updates the expression for the last range fragment to `VALUES < new`, where *new* is the new transition value:

```
INSERT INTO tabtrans VALUES (601, "BB");  -- creates interval fragment sys_p6
                                  -- with fragment expression >= 600 AND < 700
```

The fragment list and the fragment expressions for the **tabtrans** table become as follows

```
  p0        VALUES < 100                   - range fragment
  p1        VALUES < 200                   - range fragment
  p2        VALUES < 300                   - last range (or transition) fragment
  sys_p6    VALUES >= 600 AND VALUES < 700 - interval fragment
```

Here the system-generated name of the new interval fragment is **sys_p6** because `6` is the **sysfragments.evalpos** value for the new fragment in the system catalog. The **evalpos** values `7` and `5` are reserved for (not yet created) interval fragments to store rows whose fragment key values match the fragment expressions `VALUES >= 300 AND VALUES < 400` and `VALUES >= 400 AND`

`VALUES < 500`, based on the current transition value of the table and on the `INTERVAL (100)` specification in the FRAGMENT BY clause that defined the fragmentation scheme of the table.

During a change of transition value, the fragments are modified to not cause any data movement. The following statement successfully modifies the transition value to `500`.

```
ALTER FRAGMENT ON TABLE tabtrans
   MODIFY INTERVAL TRANSITION TO 500;
```

The old transition value is `300` and the new transition value is `500`, with no interval fragments in between. The first interval fragments starts at `600`. This also means that there is no data between `300` and `500`. So the expression of the last range fragment (the transition fragment) can be updated to `VALUES < 500` without data movement. Because there are interval fragments after the new transition value, the new transition value must align at an interval fragment boundary. In the above case, the new transition value `500` aligns at an interval fragment boundary (whether or not the fragment currently exists). As result of modification, the **evalpos** value in the system catalog for interval fragments changes, and the interval fragments are renamed to adhere to the **sys_p***evalpos* naming format.

The resulting modified table has the following fragments:

```
  p0     VALUES < 100  -- range fragment
  p1     VALUES < 200  -- range fragment
  p2     VALUES < 500  -- last range fragment (= transition fragment
                       -- with its expression modified)
  sys_p4 VALUES >= 600 AND VALUES < 700 - interval fragment (renamed
                       -- to sys_p4 as evalpos changes from 6 to 4
                       -- after the transition fragment change)
```

The following modification fails with an error, because there are interval fragments beyond the new transition value, and the new transition value does not align at an interval fragment boundary:

```
   ALTER FRAGMENT ON TABLE tab MODIFY INTERVAL TRANSITION TO 550;
```

The possible interval fragments are `300` to `400`, `400` to `500`, `500` to `600`, `600` to `700` and so on. The new transition value of `550` is not at an interval fragment boundary, and therefore the error is issued.

If there are interval fragments between the new and old transition value, the new transition value must align to the boundary of an interval fragment (and the interval fragment need not exist), unless the new transition value is beyond the range of the last interval fragment. All interval fragments between the new and the old transition values are converted to range fragments, and their expressions are modified to match the expression format of range fragments. The expression for the last interval fragment that was converted to a range fragment is updated to `VALUES < new`, where *new* is the new transition value.

Here is another example of INSERT operations that result in new interval fragments:

```
CREATE TABLE tab (i INT, c CHAR(2))
   FRAGMENT BY RANGE (i)
      INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
         PARTITION p0 VALUES < 100 IN dbs0,
         PARTITION p1 VALUES < 200 IN dbs1,
         PARTITION p2 VALUES < 300 IN dbs0; -- last range fragment
                                           -- or transition fragment

INSERT INTO tab
```

```
   VALUES (301, "AA"); -- creates interval fragment sys_p3 with
                       -- fragment expression >= 300 AND < 400
INSERT INTO tab
   VALUES (601, "BB"); -- creates interval fragment sys_p6
                       -- with fragment expression >= 600 AND < 700
```

The fragment list for the table now consists of these fragments:

```
  p0        VALUES < 100     -- range fragment
  p1        VALUES < 200     -- range fragment
  p2        VALUES < 300     -- range fragment
  sys_p3    VALUES >= 300 AND VALUES < 400 -- interval fragment
  sys_p6    VALUES >= 600 AND VALUES < 700 -- interval fragment
```

The ALTER FRAGMENT examples that follow are based on the above statements.

The following statement increases the transition value from `300` to `500`:

```
   ALTER FRAGMENT ON TABLE tab MODIFY INTERVAL TRANSITION TO 500;
```

Because there is an interval fragment (**sys_p3**) between the old and new transition values, that fragment is converted to a range fragment (expression becomes `< 400`). Because there is also an interval fragment (**sys_p6**) beyond the new transition value, the new transition value must align at an interval fragment boundary, which as an integer multiple of the `INTERVAL(100)` specification, it does. Here the possible interval fragments are `300` to `400`, `400` to `500`, `500` to `600`, `600` to `700` and so on. And the new transition value of `500` is at an interval fragment boundary (whose fragment need not exist). We also do not want to move data during the transition fragment modification or create any fragments. This can be accomplished by converting fragment **sys_p3** to the new transition fragment, updating its expression to `< 500` (because it is now a range fragment), and renaming it.

The resulting fragment list for the table consists of these fragments:

```
  p0          VALUES < 100  -- range fragment
  p1          VALUES < 200  -- range fragment
  p2          VALUES < 300  -- range fragment (was the old transition fragment)
  sys_p3rg    VALUES < 500  -- range fragment (was previously interval
                            -- fragment sys_p3. Its expression was modified to a
                            -- range expression. Its name was changed to a
                            -- system-generated name in format sys_p<evalpos>rq )
                            -- becomes the new transition fragment
  sys_p5      VALUES >= 600 AND VALUES < 700
                            -- interval fragment (renamed to sys_5 brcause the
                            -- evalpos value changes from 6 to 5 after the
                            -- transition fragment change.)
```

The following attempted modification of the transition value returns an error:

```
ALTER FRAGMENT ON TABLE tab
   MODIFY INTERVAL TRANSITION TO 550;
```

The statement above fails because there is an interval fragment **sys_p6** beyond the new transition value, and because the new transition value is not aligned at an interval fragment boundary.

The next example increases the transition value from `500` to `700`:

```
ALTER FRAGMENT ON TABLE tab
   MODIFY INTERVAL TRANSITION TO 700;
```

The resulting fragment list for the table includes the following fragments:

```
  p0         VALUES < 100 -- range fragment
  p1         VALUES < 200 -- range fragment
  p2         VALUES < 300 -- range fragment (was the old transition fragment)
  sys_p3rg   VALUES < 400 -- range fragment (was previously interval fragment
                          -- sys_p3, and its expression changed to a range
                            expression.
                          -- The fragment has been renamed to system-generated name
                          -- in the format sys_p<evalpos>rg ).
  sys_p6rg   VALUES < 700 -- range fragment (was previously the interval
                          -- fragment sys_p6. Its expression was modified to a
                          -- range expression and its name replaced by a system-
                          -- generated name in the format sys_p<evalpos>rg )
                          -- becomes the new transition fragment.
```

The following example increases the transition value from `700` to `750`:

```
ALTER FRAGMENT ON TABLE tab MODIFY INTERVAL TRANSITION TO 750;
```

Because there are no interval fragments beyond the new transition value, it need not align to an interval fragment boundary.

The resulting table includes the following fragments:

```
  p0         VALUES < 100  -- range fragment
  p1         VALUES < 200  -- range fragment
  p2         VALUES < 300  -- range fragment (was the old transition fragment)
  sys_p3rg   VALUES < 400  -- range fragment (was previously interval
                           -- fragment sys_p3. expression modified to a
                           -- range expression. Fragment was renamed to a system
                           -- generated name in the format sys_p<evalpos>rg)
  sys_p6rg   VALUES < 750  -- range fragment (was previously the interval
                           -- fragment sys_p6. Its expression was modified to a
                           -- range expression, and the fragment was renamed to a
                           -- system-generated name in format sys_p<evalpos>rg)
                           -- becomes the new transition fragment
```

If you wish to avoid having existing fragments automatically renamed during ALTER FRAGMENT MODIFY INTERVAL TRANSITION operations, you can first use the ALTER FRAGMENT MODIFY statement to rename with user-defined names the interval fragments whose system-generated names might otherwise be changed by the ALTER FRAGMENT MODIFY INTERVAL TRANSITION statement. The database server renames only system-generated interval fragment names (to avoid non-unique fragment names resulting when new interval fragments are created).

Using the ONLINE keyword in MODIFY operations

The ONLINE keyword instructs the database server to commit the ALTER FRAGMENT . . . MODIFY work internally, if there are no errors, and to apply an intent exclusive lock to the table, rather than an exclusive lock.

Requirements for ONLINE MODIFY operations

You can use the MODIFY option to the ALTER FRAGMENT ONLINE ON TABLE statement only for a table that is fragmented by a range interval fragmentation scheme.

Only the transition value (the starting value for interval fragments) can be modified ONLINE. All other restrictions that apply to the MODIFY option also apply to ONLINE MODIFY operations. For those restrictions, see General Restrictions for the ATTACH Clause on page 29 and Restrictions on the MODIFY clause for range interval fragments on page 56.

Example of ALTER FRAGMENT ONLINE . . . MODIFY

The following SQL statements define a fragmented **employee** table that uses a range-interval storage distribution scheme, with a unique index **employee_id_idx** on the column **emp_id** (that is also the fragmentation key) and another index **employee_dept_idx** on the column **dept_id**.

```
CREATE TABLE employee
     (emp_id INTEGER, name CHAR(32),
      dept_id CHAR(2), mgr_id INTEGER, ssn CHAR(12))
  FRAGMENT BY RANGE (emp_id)
    INTERVAL (100) STORE IN (dbs1, dbs2, dbs3, dbs4)
      PARTITION p0 VALUES < 200 IN dbs1,
      PARTITION p1 VALUES < 400 IN dbs2;
CREATE UNIQUE INDEX employee_id_idx ON employee(emp_id);
CREATE INDEX employee_dept_idx ON employee(dept_id);

INSERT INTO employee VALUES (401, "Susan", "DV", 101, "123-45-6789");
INSERT INTO employee VALUES (601, "David", "QA", 104, "987-65-4321");
```

The last two statements insert rows with fragment key values above the upper limit of the transition fragment, causing the database server to generate two new interval fragments, so that the fragment list consists of four fragments:

```
Fragments in surviving table before ALTER FRAGMENT ONLINE:
p0     VALUES < 200                   - range fragment
p1     VALUES < 400                   - range fragment (transition fragment)
sys_p2 VALUES >= 400 AND VALUES < 500 - interval fragment
sys_p4 VALUES >= 600 AND VALUES < 700 - interval fragment
```

The following statement returns an error because a transition value can only be increased. This is also a restriction for offline ALTER FRAGMENT . . . MODIFY operations.

```
ALTER FRAGMENT ONLINE ON TABLE employee
   MODIFY INTERVAL TRANSITION TO 300;
```

The following statement runs successfully:

```
ALTER FRAGMENT ONLINE ON TABLE employee MODIFY INTERVAL TRANSITION TO 600;

Fragments in surviving table after ALTER FRAGMENT ONLINE:
p0       VALUES < 200                   - range fragment
p1       VALUES < 400                   - range fragment
sys_p2rg VALUES < 600                   - range fragment (new transition fragment)
sys_p3   VALUES >= 600 AND VALUES < 700 - interval fragment
```

The following examples are also valid:

```
ALTER FRAGMENT ONLINE ON TABLE employee MODIFY INTERVAL TRANSITION TO 700;
ALTER FRAGMENT ONLINE ON TABLE employee MODIFY INTERVAL TRANSITION TO 900;
```

Examples of the MODIFY clause with interval fragments

Sections that follow illustrate syntax features of the MODIFY clause of the ALTER FRAGMENT statement, and restrictions on what the MODIFY clause can change, for tables that use range and interval fragments as their distribution strategy.

For similar examples of using the MODIFY clause with tables that are fragmented by list, see Examples of the MODIFY clause for list fragments on page 79.

Enabling or disabling range interval fragmentation

This statement disables range interval fragment creation:

```
ALTER FRAGMENT ON TABLE tab MODIFY INTERVAL DISABLED;
```

The following statement restores range interval fragment creation, undoing the effects of the previous example:

```
ALTER FRAGMENT ON TABLE tab MODIFY INTERVAL ENABLED;
```

The following statement disables range interval fragment creation, and also modifies the list of dbspaces in the STORE IN clause where new fragments will be stored (if a subsequent ALTER FRAGMENT MODIFY statement enables range interval fragment creation for table **tab**).

```
ALTER FRAGMENT ON TABLE tab MODIFY INTERVAL DISABLED
   STORE IN (dbs4, dbs5);
```

Renaming fragments in range interval fragmentation

This statement renames two range interval fragments. No IN clause specifies new storage locations, so the new names replace the existing names for the two fragments:

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p1 TO PARTITION newp1,
   PARTITION sys_p6 TO PARTITION newsys_p6;
```

The PARTITION keywords are required for range interval fragments. If you use the MODIFY clause to rename an existing fragment, the new name that you declare in the MODIFY clause cannot begin with the character string `sys`, which is reserved for system-defined fragments, but the example above successfully renames the system-defined fragment **sys_p6**.

Relocating a range or interval fragment

Suppose that the following table was created with range interval fragmentation and received two rows from insert operations:

```
CREATE TABLE tab2 (i INT, c CHAR(2))
   FRAGMENT BY RANGE (i)
   INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
      PARTITION p0 VALUES < 100 IN dbs0,
      PARTITION p1 VALUES < 200 IN dbs1;

INSERT INTO tab2 VALUES (201, "AA");
```

```
   -- creates a system-generated interval fragment sys_p2
   -- with fragment expression >= 200 AND < 300
   -- assume that this fragment is created in dbs1

INSERT INTO tab2 VALUES (601, "BB");
   -- creates a system-generated interval fragment sys_p6
   -- with fragment expression >= 600 AND < 700
   ---assume that this fragment is created in dbs2
```

The following statement instructs the database server to move range fragment **p1** from `dbs1` to `dbs2`:

```
ALTER FRAGMENT ON TABLE tab2 MODIFY
   PARTITION p1 TO PARTITION p1 IN dbs2;
```

The next example moves range fragment **p1** from `dbs1` to `dbs2` and moves interval fragment **sys_p6** from `dbs2` to `dbs3`:

```
ALTER FRAGMENT ON TABLE tab2 MODIFY
   PARTITION p1 TO PARTITION p1 IN dbs2,
   PARTITION sys_p6 TO PARTITION sys_p6 IN dbs3;
```

Replacing the list of dbspaces that store new interval fragments

The following CREATE TABLE statement defines a range interval fragmentation strategy, in which

- column **i** is the fragmentation key,
- 100 is the range interval size,
- new fragments will be stored in dbspaces `dbs1`, `dbs2`, and `dbs3`,
- the initial fragments **p0** (in dbspace `dbs0`) and **p1** (in dbspace `dbs1`) have transition values of `100` and `200` respectively.

```
CREATE TABLE tab (i INT, c CHAR(2))
   FRAGMENT BY RANGE (i)
   INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
      PARTITION p0 VALUES < 100 IN dbs0,
      PARTITION p1 VALUES < 200 IN dbs1;
```

The following ALTER FRAGMENT statement replaces the STORE IN list (`dbs1`, `dbs2`, `dbs3`) with a new list (`dbs4`, `dbs5`).

```
ALTER FRAGMENT ON TABLE tab
   MODIFY INTERVAL STORE IN (dbs4, dbs5);
```

In the example above, the MODIFY clause specifies that new fragments will be created alternately in `dbs4` and `dbs5`. Any system-defined fragments (and the fragment **p1**) that were created in dbspaces of the original STORE IN list (`dbs1`, `dbs2`, `dbs3`) remain in those dbspaces. Existing and subsequently inserted rows whose fragmentation-key values are within the range intervals of those fragments continue to be stored in those fragments, but new interval fragments will be created, alternating in round-robin fashion, in the `dbs4` and `dbs5` dbspaces.

Consider that the following fragmented table:

```
CREATE TABLE mytab (col1 int)
   FRAGMENT BY RANGE (c1) INTERVAL (100)
   STORE IN (dbs1, dbs2, dbs3, dbs4, dbs5)
      PARTITION p1 VALUES < 300 in dbs0;
```

This ALTER FRAGMENT statement replaces the list of dbspaces where new interval fragments will be stored:

```
ALTER FRAGMENT ON TABLE mytab MODIFY
   STORE IN (dbs1, dbs6, dbs3, dbs4, dbs8);
```

The new list replaces `dbs2` with `dbs6` and replaces `dbs5` with `dbs8`. If you want any of the dbspaces from the current STORE IN list to be available for new fragments, the MODIFY clause must also include them in the new list, which replaces the old list in the modified fragmentation scheme. In the example above, new range interval fragments will be created in the five dbspaces listed after the STORE IN keywords, but any existing fragments that were created in **dbs2** and **dbs5** continue to store rows whose data values match the fragmentation key ranges for those fragments.

You can modify the list of dbspaces in the STORE IN clause. The old list is replaced by the new list that you specify. Existing fragments in the old dbspace are not moved. Consider the following table:

You can move an existing fragment to another dbspace by changing the IN *dbspace* specification for the fragment:

```
CREATE TABLE tab (i INT, c CHAR(2))
   FRAGMENT BY RANGE (i)
   INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
      PARTITION p0 VALUES < 100 IN dbs0,
      PARTITION p1 VALUES < 200 IN dbs1;

INSERT INTO tab VALUES (201, "AA");
   -- creates interval fragment sys_p2
   -- with fragment expression >= 200 AND < 300
   -- (assume that this fragment is created in dbs1)
INSERT INTO tab VALUES (601, "BB");
   -- creates interval fragment sys_p6
   -- with fragment expression >= 600 AND < 700
   -- (assume that this fragment is created in dbs2)
```

The next statement instructs the database server to moves fragment **p1** from **dbs1** to **dbs2**:

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p1 TO PARTITION p1 IN dbs2;
```

The next example moves range fragment **p1** from **dbs1** to **dbs2**, and moves interval fragment **sys_p6** from **dbs2** to **dbs3**;

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p1 TO PARTITION p1 IN dbs2,
   PARTITION sys_p6 TO PARTITION sys_p6 IN dbs3;
```

You cannot, however, modify an expression for an interval fragment after the system has generated the fragment. Consider this table:

```
CREATE TABLE tab (i INT, c CHAR(2))
   FRAGMENT BY RANGE (i)
   INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
      PARTITION p0 VALUES < 100 IN dbs0,
      PARTITION p1 VALUES < 200 IN dbs1;

INSERT INTO tab VALUES (201, "AA");
   -- creates interval fragment sys_p2
   -- with fragment expression >= 200 AND < 300
INSERT INTO tab VALUES (601, "BB");
   -- creates interval fragment sys_p6
   -- with fragment expression >= 600 AND < 700
```

Now you cannot modify the fragment expression for **sys_p2** or **sys_p6**. An error is returned if you try to do so.

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION sys_p6 TO PARTITION sys_p6
      VALUES < 900 IN dbs2;
```

The above statement fails with an error.

Modifying the expression that defines a range fragment

Under some circumstances, you can use the MODIFY clause to change the expression that defines a range fragment; examples that follow illustrate various restrictions on changes that you can make to the expressions that define to range fragments. You cannot, however, modify an expression for an interval fragment after the system has generated that fragment. Consider this table:

```
CREATE TABLE tab (i INT, c CHAR(2))
   FRAGMENT BY RANGE (i)
   INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
      PARTITION p0 VALUES < 100 IN dbs0,
      PARTITION p1 VALUES < 200 IN dbs1;

INSERT INTO tab VALUES (201, "AA");
   -- creates interval fragment sys_p2
   -- with fragment expression >= 200 AND < 300
INSERT INTO tab VALUES (601, "BB");
   -- creates interval fragment sys_p6
    -- with fragment expression >= 600 AND < 700
```

Now you cannot modify the expressions for interval fragments **sys_p2** or **sys_p6**. The database server returns an error if you try to do so.

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION sys_p6 TO PARTITION sys_p6
      VALUES < 900 IN dbs2;
```

The above statement fails with an error.

You can modify an expression for the first intermediate range fragment, but the replacement expression cannot cross adjacent fragment boundaries. This operation can result in data movement. Here is an example;

```
CREATE TABLE tab (i INT, c CHAR(2))
   FRAGMENT BY RANGE (i)
   INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
   PARTITION p0 VALUES < 100 IN dbs0,
   PARTITION p1 VALUES < 200 IN dbs1,
   PARTITION p2 VALUES < 300 IN dbs0;

INSERT INTO tab VALUES (301, "AA");
   -- creates interval fragment sys_p3
   -- with fragment expression >= 300 AND < 400
INSERT INTO tab VALUES (601, "BB");
   -- creates interval fragment sys_p6
   -- with fragment expression >= 600 AND < 700
```

All of the ALTER examples below are based on fragments of the table defined in the CREATE statement above. The following ALTER FRAGMENT statement modifies the expression for range fragment **p0**

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p0 TO PARTITION p0
      VALUES < -50 IN dbs0;
```

The following modifies expression for fragment **p0** and also moves the fragment from `dbs0` to `dbs5`

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p0 TO PARTITION p0
      VALUES < -50 IN dbs5;
```

The following statement successfully makes three changes to fragment **p0**:

- modifies the fragment expression for **p0**,
- modifies the fragment name to **newp0**,
- and also moves the renamed fragment from `dbs0` to `dbs5`.

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p0 TO PARTITION newp0
      VALUES < -50 IN dbs5;
```

The next example fails with an error, however, because the new expression for fragment **p0** crosses the boundary of the range of the next adjacent fragment **p1**

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p0 TO PARTITION p0
      VALUES < 250 IN dbs0;
```

The following ALTER FRAGMENT example successfully modifies the expression for range fragment **p1**:

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p1 TO PARTITION p1
      VALUES < 150 IN dbs1;
```

The following modification fails with an error, because the new expression for fragment **p1** crosses the boundary of the previous adjacent fragment **p0**:

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p0 TO PARTITION p0
      VALUES < 50 IN dbs0;
```

If for any reason, as result of the ALTER FRAGMENT MODIFY operation, the rows cannot be moved to the new fragments, an error is returned. This is an example:

```
CREATE TABLE tab (i INT, c CHAR(2))
   FRAGMENT BY RANGE (i)
      INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
      PARTITION p0 VALUES IS NULL IN dbs0,
      PARTITION p1 VALUES < 200 IN dbs1,
      PARTITION p2 VALUES < 300 IN dbs0;

ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p0 TO PARTITION p0
      VALUES < 100 IN dbs0;
```

As a result of the modification, the resultant table will have the following fragments

```
PARTITION p0 VALUES < 100 IN dbs0,
      PARTITION p1 VALUES < 200 IN dbs1,
      PARTITION p2 VALUES < 300 IN dbs0
```

If the previous NULL fragment stored any rows (meaning rows in the table that have NULL value for column **i**), then those rows would not fit in any of the fragments in the new fragmentation scheme. The above ALTER FRAGMENT operation would therefore fail while moving rows.

Also note that the NULL fragment is always the first fragment in the table. Even if the user specifies the NULL fragment as the last fragment during CREATE TABLE or ALTER TABLE operations, it is rearranged as the first fragment in the table, with the smallest **evalpos** value in the fragment list. While modifying first and intermediate range fragments, the database server imposes the restriction that the new expression cannot cross adjacent fragment boundaries. So while modifying the NULL fragment, whatever new expression you specify cannot cross the boundary of the next range or interval fragment. Here is an example:

```
CREATE TABLE tab (i INT, c CHAR(2))
   FRAGMENT BY RANGE (i)
   INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
      PARTITION p0 VALUES IS NULL IN dbs0,
      PARTITION p1 VALUES < 200 IN dbs1,
      PARTITION p2 VALUES < 300 IN dbs0;
```

Suppose that the table does not have any rows in fragment **p0**. In this case, **p0** can be modified to a non-NULL fragment.

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p0 TO PARTITION p0
      VALUES < 250 IN dbs0;
```

Because the new expression for **p0** (`VALUES < 250`) crosses the boundary for **p1** (`VALUES < 200`), however, the example above returns an error.

The following ALTER FRAGMENT statement is possible:

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p0 TO PARTITION p0 VALUES < 150 IN dbs0;
```

You can modify expression for last range fragment (transition fragment) but you can only increase the transition value. There is no data movement in this operation.

```
CREATE TABLE tab (i INT, c CHAR(2))
   FRAGMENT BY RANGE (i)
   INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
      PARTITION p0 VALUES < 100 IN dbs0,
      PARTITION p1 VALUES < 200 IN dbs1,
      PARTITION p2 VALUES < 300 IN dbs0;
         -- last range fragment or transition fragment
```

The following modification returns an error, because it attempts to decrease the transition value (from 300 to 250):

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p2 TO PARTITION p2
      VALUES < 250 IN dbs0;
```

The following statement modifies the fragment expression for **p2** (the transition fragment). Because there are not yet any system-generated interval fragments, the new transition value need not align at the interval fragment boundary.

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p2 TO PARTITION p2
      VALUES < 350 IN dbs0;
```

If there are no interval fragments between new and the old transition value, you can update the expression for the last range fragment to `VALUES <` *new transition value*. Here is an example:

```
CREATE TABLE tab (i INT, c CHAR(2))
   FRAGMENT BY RANGE (i)
   INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
       PARTITION p0 VALUES < 100 IN dbs0,
       PARTITION p1 VALUES < 200 IN dbs1,
       PARTITION p2 VALUES < 300 IN dbs0;
         -- last range fragment is the "transition fragment"

INSERT INTO tab VALUES (601, "BB");
   -- creates interval fragment sys_p6
   -- with fragment expression >= 600 AND < 700
   -- (assume that this fragment is created in dbs3)
```

The modified table now has these fragments:

Fragment

Expression and Fragment Type

p0

`VALUES < 100` – range fragment

p1

`VALUES < 200` – range fragment

p2

`VALUES < 300` - last range fragment (or transition fragment)

sys_p6

`VALUES >= 600 AND VALUES < 700` - interval fragment

During the change of transition value, the fragments are modified in a manner that does not cause any data movement.

The following statement modifies the fragment expression for **p2**, the transition fragment (or last range fragment).

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p2 TO PARTITION p2 VALUES < 500 IN dbs0;
```

The old transition value was 300 and the new transition value is 500. There are no system-generated interval fragments between these range fragments, and the first interval fragments starts at 600. This also means that there is no data rows between 300 and 500, so the expression of the transition fragment (the last range fragment) can be updated to `VALUES < 500` without data movement. Because there are now interval fragments after the new transition value, the new transition value must align at an interval fragment boundary. In this case, the new transition value of 500 aligns with an interval fragment boundary. (The interval fragment need not exist.)

As a result of this modification, the **evalpos** value for subsequent interval fragments changes, and interval fragments are renamed to adhere to the format for system-generated fragment names. After this ALTER TABLE MODIFY operation, the resulting table has these fragments:

Fragment

Expression and Fragment Type

p0

`VALUES < 100` – range fragment

p1

`VALUES < 200` – range fragment

p2

`VALUES < 500` – modified expression for transition fragment

sys_p4

`VALUES >= 600 AND VALUES <700` – interval fragment

Here the modified expression is for fragment **p2**. which is the last range fragment. (This is also called the *transition fragment*, because any fragments to store larger values in the range of the fragment key will be system-generated interval fragments.) The system-generated interval fragment is renamed to **sys_p4** because the **evalpos** value changes from 6 to 4 after the expression for the transition fragment changed.

The following modification fails with an error because there are interval fragments beyond the new transition value, and the new transition value does not align at an interval fragment boundary:

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p2 TO PARTITION p2 VALUES YY 550 IN dbs0;
```

The ranges of the possible interval fragments are 300 to 400, 400 to 500, 500 to 600, 600 to 700, and so on, but the new transition value of 550 is not at an interval fragment boundary, and so the database server issues an error.

If there are interval fragments between the new and old transition value, then the new transition value must align to the boundary of an interval fragment (but that interval fragment need not exist), unless the new transition value is beyond the last interval fragment. All interval fragments between the new and old transition values are converted to range fragments, and their expressions are modified to match the range fragment expressions. The expression for the last interval fragment that was converted to a range fragment is updated to `VALUES <` *new transition value*.

This behavior is illustrated by the following example

```
CREATE TABLE tab (i INT, c CHAR(2))
   FRAGMENT BY RANGE (i)
   INTERVAL (100) STORE IN (dbs1, dbs2, dbs3)
      PARTITION p0 VALUES < 100 IN dbs0,
      PARTITION p1 VALUES < 200 IN dbs1,
      PARTITION p2 VALUES < 300 IN dbs0;
        -- last range fragment or transition fragment

INSERT INTO tab VALUES (301, "AA");
   -- creates interval fragment sys_p3
   -- with fragment expression >= 300 AND < 400
   -- (assume this fragment is created in dbs1)
INSERT INTO tab VALUES (601, "BB");
   -- creates interval fragment sys_p6
```

```
   -- with fragment expression >= 600 AND < 700
   -- (assume this fragment is created in dbs3)
```

After the two INSERT operations, the table would have these range and interval fragments:

Fragment

Expression and Fragment Type

p0

`VALUES < 100` – range fragment

p1

`VALUES < 200` – range fragment

p2

`VALUES < 300` – range fragment

sys_p3

`VALUES >= 300 AND VALUES <400` – interval fragment

sys_p4

`VALUES >= 600 AND VALUES <700` – interval fragment

The ALTER FRAGMENT examples that follow are based on this table.

The following example modifies the expression for fragment **p2** (the transition fragment).

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p2 TO PARTITION p2 VALUES < 500 IN dbs0;
```

Because there is an interval fragment (**sys_p3**) between the old and new transition value, that fragment is converted to range fragment (whose expression becomes `VALUES < 400`).

And because there are interval fragments beyond the new transition value (for fragment **sys_p6**), the new transition value must align at an interval fragment boundary, which it does. The possible interval fragments must be an integer multiple of the range interval size (including 400 to 500, 500 to 600, 700 to 800, and so on. The new transition value is 500, which is at an interval fragment boundary. It is also efficient to avoid move data during the transition fragment modification, and to avoid creating any fragments. This can be made possible by converting fragment **sys_p3** to the new transition fragment, updating its expression to `< 500`, and renaming it to the name of the old transition fragment.

The resulting table has the following fragments:

Fragment

Expression and Fragment Type

p0

VALUES < 100 – range fragment

p1

VALUES < 200 – range fragment

sys_p2rg

VALUES < 300 – range fragment (This was the old transition fragment, now renamed **sys_p2rg** in the system generated format **sys_p***evalpos***rg**.)

p2

VALUES <500 - range fragment (This was previously interval fragment **sys_p3**. Its expression, modified to a range expression. now defines the new transition fragment)

sys_p5

VALUES >= 600 AND VALUES <700 – interval fragment (renamed to **sys_5** as its **evalpos** value changes from 6 to 5 after the transition fragment change)

The following modification of transition fragment **p2** returns an error:

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p2 TO PARTITION p2 VALUES < 550 IN dbs0;
```

The error is issued because there is an interval fragment **sys_p6** beyond the new transition value, and the new transition value is not aligned at an interval fragment boundary.

The next example modifies the expression for fragment **p2**, which is a transition fragment:

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p2 TO PARTITION p2 VALUES < 750 IN dbs0;
```

Because there are no interval fragments beyond the new transition value, it need not align to an interval fragment boundary.

The resulting table has the following fragments:

Fragment

Expression and Fragment Type

p0

`VALUES < 100` – range fragment

p1

`VALUES < 200` – range fragment

sys_p2rg

`VALUES < 300` – range fragment (This was the old transition fragment, now renamed **sys_p2rg** in the system generated format **sys_p***evalpos***rg**.)

sys_p3rg

`< 400` – range fragment (This was previously interval fragment **sys_p3** before its expression was modified to a range expression.)

p2

`VALUES <750` - range fragment (Previously interval fragment **sys_p6** before its expression was modified to a range expression. This becomes the new transition fragment.)

Examples of the MODIFY clause for list fragments

You can use the MODIFY clause to change fragments of a table or index that is fragmented by list, including these modifications:

- Change the names of existing list fragments
- Move the storage location of an existing list fragment to another dbspace
- Modify the expression list for one or more list fragments.

The following ALTER FRAGMENT ON TABLE statement modifies the name, the list of fragment expressions, and the storage location for a fragment of a table that is partitioned by list:

```
ALTER FRAGMENT ON TABLE T2 MODIFY
   PARTITION part1 TO PARTITION part11
      VALUES ('CA', 'OR', 'TX') IN dbs1;
```

This changes the partition name from **part1** to **part11**, adds the value `'TX'` to the list of expressions for that fragment, and moves the renamed partition into the dbspace **dbs1**.

Examples that follow illustrate these and other uses of the MODIFY clause with list fragmentation schemes, and also illustrate MODIFY operations that fail because of logical restrictions on list fragmentation.

Suppose that this CREATE TABLE statement has defined table **myTable** with the following schema and with a list fragmentation strategy:

```
CREATE TABLE myTable (i int, c char(2))
    FRAGMENT BY LIST (c)
        PARTITION p1 VALUES ("AB", "CD") IN dbs1,
        PARTITION p2 VALUES ("PQ", "RS") IN dbs2,
        PARTITION p3 REMAINDER IN dbs3;
```

The next ALTER FRAGMENT statement modifies the storage distribution strategy for the **p2** fragment:

```
ALTER FRAGMENT ON TABLE myTable MODIFY
   PARTITION p2 TO PARTITION newp2
      VALUES (NULL) IN dbs5;
```

The statement above has these effects on the definition of the fragment and its storage distribution:

- redefines the fragment expression for fragment **p2** to make it a NULL fragment,
- changes the fragment name to **newp2**,
- moves the storage location of that fragment from `dbs2` to `dbs5`,
- and moves any existing data rows that had been stored in fragment **p2** to the remainder fragment **p3**, because the fragment key values (`"PQ"` and `"RS"`) in column **c** of those rows do not match the new NULL expression.

If the automatic mode for updating distribution statistics is enabled, an ALTER FRAGMENT . . . MODIFY statement that results in data redistribution causes the fragment-level statistics for the affected fragments to be dropped. Table-level statistics, however, are not dropped. Because no fragment-level statistics exist for the affected fragments, the next explicit or

automatic UPDATE STATISTICS operation on the table will rebuild the fragment-level distributions, and store the results in the system catalog.

The ALTER FRAGMENT examples that follow specify modifications to fragments of the **tab** table that this CREATE TABLE statement defines with a list fragmentation scheme:

```
CREATE TABLE tab (i int, c char(2))
    FRAGMENT BY LIST (c)
        PARTITION p1 VALUES ("AB", "CD") IN dbs1,
        PARTITION p2 VALUES ("PQ", "RS") IN dbs2,
        PARTITION p3 VALUES (NULL) IN dbs3,
        PARTITION p4 REMAINDER IN dbs4;
```

The following modifies fragment expression for fragment **p1**

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p1 TO PARTITION p1
      VALUES ("AB", "CD", "EF") IN dbs1;
```

The following statement modifies the fragment expression for fragment **p3**:

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p3 TO PARTITION p3
      VALUES ("XX", "YY", "ZZ") IN dbs3;
```

If for any reason, as result of the ALTER FRAGMENT ON TABLE MODIFY operation, any rows cannot be moved to the new fragments, an error is returned, as in the following example:

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p3 TO PARTITION p3
      VALUES ("XX", "YY", "ZZ") IN dbs3;
```

As a result of the modification, the resulting storage distribution scheme for the **tab** table will have the following fragments:

```
    PARTITION p1 VALUES ("AB", "CD")       IN dbs1,
    PARTITION p2 VALUES ("PQ", "RS")       IN dbs2,
    PARTITION p3 VALUES ("XX", "YY", "ZZ") IN dbs2
```

If the previous remainder fragment **p3** had a row with value `"AA"` in column c, then that row does not fit in any of the fragments in the new fragmentation scheme. The ALTER FRAGMENT statement above would therefore fail with an error while attempting to move rows from the remainder fragment.

The next three examples illustrate modifications to the same table fragmentation scheme that will fail because of overlaps.

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p2 TO PARTITION p2 VALUES (NULL) IN dbs2;
```

Because the ALTER FRAGMENT statement above attempts to change fragment **p2** into a duplicate NULL fragment, the statement fails with an error, because the NULL fragment **p3** already exists.

The following modification of the same table attempts to modify fragment **p2** into a duplicate remainder fragment:

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p2 TO PARTITION p2 REMAINDER IN dbs2;
```

The statement above fails with an error, because the existing fragment **p4** is already defined as a remainder fragment.

The following modification creates a duplicate expression-list value `"RS"` in two of the fragments:

```
ALTER FRAGMENT ON TABLE tab MODIFY
   PARTITION p1 TO PARTITION p1
      VALUES ("AB", "CD", "RS") IN dbs1;
```

Because list value `"RS"` is already defined in the expression list for fragment **p2**, the statement above fails with an error.

For an example of using the MODIFY option to the ALTER FRAGMENT ON INDEX statement, see Examples of ALTER FRAGMENT ON INDEX statements on page 81.

Examples of ALTER FRAGMENT ON INDEX statements

The following series of examples illustrate the use of ALTER FRAGMENT ON INDEX with the INIT, ADD, DROP, and MODIFY options.

This first example creates an index stored in **dbsp1**:

```
CREATE INDEX item_idx ON items (stock_num) IN dbsp1;
```

The following statement modifies the index to add fragmentation by expression. Values up to 50 are stored in **dbsp1**, values between 51 and 80 in **dbsp2**, and the remainder in **dbsp3**:

```
ALTER FRAGMENT ON INDEX item_idx INIT
   FRAGMENT BY EXPRESSION
   stock_num <= 50 IN dbsp1,
   stock_num > 50 AND stock_num <= 80 IN dbsp2,
   REMAINDER IN dbsp3;
```

The following statement adds a new fragment to the index:

```
ALTER FRAGMENT ON INDEX item_idx
   ADD stock_num > 80 AND stock_num <= 120 IN dbsp4;
```

The following statement changes the first fragment of the index:

```
ALTER FRAGMENT ON INDEX item_idx
   MODIFY dbsp1 TO stock_num <= 40 IN dbsp1;
```

The following statement drops the fragment in **dbsp4** from the index:

```
ALTER FRAGMENT ON INDEX item_idx
   DROP dbsp4;
```

The following statement defines an index that is fragmented by expression, with the fragments stored in named partitions of the dbspaces **dbsp1** and **dbsp2**:

```
ALTER FRAGMENT ON INDEX item_idx INIT
   PARTITION BY EXPRESSION
   PARTITION part1 stock_num <= 10 IN dbsp1,
   PARTITION part2 stock_num > 20 AND stock_num <= 30 IN dbsp1,
   PARTITION part3 REMAINDER IN dbsp2;
```

The following statement adds a new named fragment:

```
ALTER FRAGMENT ON INDEX item_idx ADD
   PARTITION part4 stock_num > 30 AND stock_num <= 40 IN dbsp2
```

```
   BEFORE part3;
```

The following statement defines a range interval storage distribution scheme on the index **idx1**:

```
ALTER FRAGMENT ON INDEX idx2 INIT
   FRAGMENT BY RANGE(c2)
      INTERVAL (NUMTOYMINTERVAL(1,'MONTH')
         PARTITION part0 VALUES <  DATE('01/01/2007') IN dbs0,
         PARTITION part1 VALUES <  DATE('07/01/2007') IN dbs1,
         PARTITION part2 VALUES <  DATE('01/01/2008') IN dbs2
```

In the example above,

- the fragmentation key is the value of column **c2**,
- the interval value is one month,
- because no STORE IN clause is included, new system-generated interval partitions will be stored in `dbs0`, `dbs1`, and `dbs2` in round-robin fashion;
- the interval partition transition value is 01/01/2008. (This is the smallest value beyond the range of the last user-defined fragment.)

The following statement defines a list storage distribution scheme on the index **idx2**:

```
ALTER FRAGMENT ON INDEX idx2 INIT
   FRAGMENT BY LIST(state)
      PARTITION part0 VALUES ('KS','IL') IN dbs0,
      PARTITION part1 VALUES ('CA','OR') IN dbs0,
      PARTITION part2 VALUES (NULL) IN dbs1,
      PARTITION part3 REMAINDER IN dbs2;
```

In the list fragmentation example above,

- the fragmentation key is the value of column **state**,
- the expression lists for the first two fragments are each the strings for postal abbreviations of two states,
- and both a NULL fragment (**part2**) and a remainder fragment (**part3**) are defined for rows with fragmentation key values that do not match the first two fragment expression lists.

ALTER FUNCTION statement

Use the ALTER FUNCTION statement to change the routine modifiers or pathname of a user-defined function. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_105_Connect_42_altfunc002) unique_105_Connect_42_altfunc002(explicit id unique_105_Connect_42_altfunc003) unique_105_Connect_42_altfunc003(explicit id unique_105_Connect_42_altfunc004) unique_105_Connect_42_altfunc004(explicit id unique_105_Connect_42_altfunc005) unique_105_Connect_42_altfunc005 **ALTER** { **FUNCTION***function* ([*parameter_type*]) | **SPECIFIC FUNCTION** <Specific Name>[] } **WITH**({ | { **ADD** | **MODIFY** | **DROP** } <Routine Modifier>[] | [] **MODIFY EXTERNAL NAME** = <Shared-Object Filename>[] })

Element	Description	Restrictions	Syntax
function	User-defined function to be modified	Must be registered in the database. If the name does not uniquely identify a function, you must enter one or more appropriate values for *parameter_type*.	Identifier on page 1231
parameter_type	Data type of a parameter	Must be the same data types (and specified in the same order) as in the definition of *function*.	Data Type on page 985

Usage

The ALTER FUNCTION statement can modify a user-defined function to tune its performance by modifying characteristics that control how the function executes. You can also add or replace related user-defined routines (UDRs) that provide alternatives for the query optimizer, which can improve performance.

All modifications take effect on the next invocation of the function.

Only the UDR owner or the DBA can use the ALTER FUNCTION statement.

Keywords That Introduce Modifications

Use the following keywords to introduce what you modify in the UDR.

Keyword	Effect on Specified Routine Modifier
ADD	Add a new routine modifier to the UDR
MODIFY	Change an attribute of the routine modifier
DROP	Delete the routine modifier from the UDR
MODIFY EXTERNAL NAME (for external functions only)	Replace the file specification of the executable file. When the IFX_EXTEND_ROLE configuration parameter = ON, this option is valid only for users to whom the DBSA has granted the EXTEND role. With IFX_EXTEND_ROLE = OFF (or not set), the UDR owner or the DBA can use this option.
WITH	Introduces all modifications

If the routine modifier is a BOOLEAN value, MODIFY sets the value to be `t` (equivalent of using the keyword ADD to add the routine modifier). For example, both of the following statements alter the **func1** function so that it can be executed in parallel in the context of a parallelizable data query:

```
ALTER FUNCTION func1 WITH (MODIFY PARALLELIZABLE);
ALTER FUNCTION func1 WITH (ADD PARALLELIZABLE);
```

See also Example of Altering Routine Modifiers on page 88.

ALTER INDEX statement

Use the ALTER INDEX statement to change the clustering attribute of an existing index. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_112_Connect_42_altix001) unique_112_Connect_42_altix001(explicit id unique_112_Connect_42_altix002) unique_112_Connect_42_altix002

```
ALTER INDEX index { TO [ NOT ] CLUSTER }
```

Element	Description	Restrictions	Syntax
index	Name of the index to be altered	Must exist	Identifier on page 1231

Usage

ALTER INDEX is valid only for indexes that the CREATE INDEX statement created explicitly. ALTER INDEX cannot modify an index on a temporary table, nor an index that the database server created implicitly to support a constraint.

You cannot change the collating order of an existing index. If you use ALTER INDEX to modify an index after the SET COLLATION statement of SQL has specified a non-default collating order, the SET COLLATION statement has no effect on the index.

The ALTER INDEX statement cannot reference a forest of trees index. For information on forest of trees indexes, see HASH ON clause on page 249.

TO CLUSTER Option

The TO CLUSTER option causes the database server to reorder the rows of the physical table according to the order of index-key values.

Clustering an index rebuilds the table in a different location within the same dbspace. When you run the ALTER INDEX statement with the `TO CLUSTER` keywords, all of the extents associated with the previous version of the table are released. The resulting newly-built version of the table has no empty extents.

For an ascending index, `TO CLUSTER` puts rows in lowest-to-highest order. For a descending index, the rows are reordered in highest-to-lowest order.

When you reorder, the entire file is rewritten. This process can take a long time, and it requires sufficient disk space to maintain two copies of the table.

While a table is clustering, it is locked `IN EXCLUSIVE MODE`. When another process is using the table to which the index name belongs, the database server cannot execute the ALTER INDEX statement with the `TO CLUSTER` keywords; it returns an error unless lock mode is set to `WAIT`. (When lock mode is set to `WAIT`, the database server retries the ALTER INDEX statement.)

Over time, if you modify the table, you can expect the benefit of an earlier cluster to disappear because rows are added in space-available order, not sequentially. You can recluster the table to regain performance by issuing another ALTER INDEX TO

CLUSTER statement on the clustered index. You do not need to drop a clustered index before you issue another ALTER INDEX TO CLUSTER statement on a currently clustered index.

Example of clustering an index

The following example shows how you can use the ALTER INDEX TO CLUSTER statement to physically order the rows in the **orders** table. The CREATE INDEX statement creates an index on the **customer_num** column of the table. Then the ALTER INDEX statement causes the physical reordering of the existing rows within the **orders** table: following

```
CREATE INDEX ix_cust ON orders (customer_num);
ALTER INDEX ix_cust TO CLUSTER;
```

TO NOT CLUSTER Option

The TO NOT CLUSTER option drops the cluster attribute on the specified index without affecting the physical order of rows in the table.

Because no more than one clustered index can exist on a given table, you must use the TO NOT CLUSTER option to release the cluster attribute from one index before you assign it to another index on the same table.

Example of dropping the clustering attribute

The following statements illustrate how to remove clustering from one index and how a second index physically reclusters the table:

```
CREATE UNIQUE INDEX ix_ord ON orders (order_num);

CREATE CLUSTER INDEX ix_cust ON orders (customer_num);
. . .
ALTER INDEX ix_cust TO NOT CLUSTER;

ALTER INDEX ix_ord TO CLUSTER;
```

The first two statements create indexes for the **orders** table and cluster the physical table in ascending order on the **customer_num** column. The last two statements recluster the physical table in ascending order on the **order_num** column.

ALTER PROCEDURE statement

Use the ALTER PROCEDURE statement to change the routine modifiers or pathname of a previously defined external procedure. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_116_Connect_42_altpro002) unique_116_Connect_42_altpro002(explicit id unique_116_Connect_42_altpro003) unique_116_Connect_42_altpro003(explicit id unique_116_Connect_42_altpro004) unique_116_Connect_42_altpro004(explicit id unique_116_Connect_42_altpro005) unique_116_Connect_42_altpro005

ALTER { **PROCEDURE***procedure*([*parameter_type*]) | **SPECIFIC PROCEDURE** <Specific Name>[] } **WITH**({ | { **ADD** | **MODIFY** | **DROP** } <Routine Modifier>[] | [] **MODIFY EXTERNAL NAME** = <Shared-Object Filename>[] })

Element	Description	Restrictions	Syntax
procedure	User-defined procedure to modify	Must be registered in the database. If the name does not uniquely identify a function, you must enter one or more appropriate values for *parameter_type*.	Identifier on page 1231
parameter_type	Data type of a parameter	Must be the same data types (and specified in the same order) as in the definition of *procedure*.	Data Type on page 985

Usage

The ALTER PROCEDURE statement enables you to modify an external procedure to tune its performance by modifying characteristics that control how it executes. You can also add or replace related UDRs that provide alternatives for the optimizer, which can improve performance. All modifications take effect on the next invocation of the procedure.

Only the UDR owner or the DBA can use the ALTER PROCEDURE statement.

If the procedure name is not unique among routines registered in the database, you must enter one or more appropriate values for *parameter_type*.

The following keywords introduce what you want to modify in *procedure*.

Keyword	Effect
ADD	Add a new routine modifier to the UDR
MODIFY	Change an attribute of a routine modifier
DROP	Delete a routine modifier from the UDR
MODIFY EXTERNAL NAME (for external procedures only)	Replace the file specification of the executable file. When the IFX_EXTEND_ROLE configuration parameter = ON, this option is valid only for users to whom the DBSA has granted the EXTEND role. With IFX_EXTEND_ROLE = OFF (or not set), the UDR owner or the DBA can use this option.
MODIFY EXTERNAL NAME (for external procedures only)	Replace the file specification of the executable file. (Valid only for users who have the EXTEND role)
WITH	Introduces all modifications

If the routine modifier is a BOOLEAN value, MODIFY sets the value to be `T` (equivalent to using the keyword ADD to add the routine modifier). For example, both of the following statements alter the **proc1** procedure so that it can be executed in parallel in the context of a parallelizable data query:

```
ALTER PROCEDURE proc1 WITH (MODIFY PARALLELIZABLE);
ALTER PROCEDURE proc1 WITH (ADD PARALLELIZABLE);
```

See also Example of Altering Routine Modifiers on page 88.

ALTER ROUTINE statement

Use the ALTER ROUTINE statement to change the routine modifiers or pathname of a previously defined user-defined routine (UDR), This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_117_Connect_42_altrou002) unique_117_Connect_42_altrou002(explicit id unique_117_Connect_42_altrou003) unique_117_Connect_42_altrou003(explicit id unique_117_Connect_42_altrou004) unique_117_Connect_42_altrou004(explicit id unique_117_Connect_42_altrou005) unique_117_Connect_42_altrou005

```
ALTER { ROUTINE routine ( [ parameter_type ] ) | SPECIFIC ROUTINE
<Specific Name> } WITH( { | { ADD | MODIFY | DROP } <Routine Modifier> | MODIFY EXTERNAL NAME = <Shared-Object
Filename> } )
```

Element	Description	Restrictions	Syntax
routine	User-defined routine to modify	Must be registered in the database. If the name does not uniquely identify a routine, you must enter one or more appropriate values for *parameter_type*.	Identifier on page 1231
parameter_type	Data type of a parameter	Must be the same data types (and specified in the same order) as in the definition of *routine*.	Data Type on page 985

Usage

The ALTER ROUTINE statement allows you to modify a previously defined UDR to tune its performance by modifying characteristics that control how the UDR executes. You can also add or replace related UDRs that provide alternatives for the optimizer, which can improve performance.

This statement is useful when you do not know whether a UDR is a user-defined function or a user-defined procedure. When you use this statement, the database server alters the appropriate user-defined procedure or user-defined function.

All modifications take effect on the next invocation of the UDR.

Only the UDR owner or the DBA can use the ALTER ROUTINE statement.

Restrictions

If the name does not uniquely identify a UDR, you must enter one or more appropriate values for *parameter_type*.

When you use this statement, the type of UDR cannot be ambiguous. The UDR that you specify must refer to either a user-defined function or a user-defined procedure. If either of the following conditions exist, the database server returns an error:

- The name (and parameters) that you specify applies to both a user-defined procedure and a user-defined function.
- The specific name that you specify applies to both a user-defined function and a user-defined procedure.

Keywords That Introduce Modifications

Use these keywords to introduce the items in the UDR that you want to modify:

Keyword	Effect
ADD	Add a routine modifier to the UDR
DROP	Delete a routine modifier from the UDR
MODIFY	Change an attribute of the routine modifier
MODIFY EXTERNAL NAME (for external routines only)	Replace the file specification of the executable file. When the IFX_EXTEND_ROLE configuration parameter = ON, this option is valid only for users to whom the DBSA has granted the EXTEND role. With IFX_EXTEND_ROLE = OFF (or not set), the UDR owner or the DBA can use this option.
WITH	Introduces all modifications

If the routine modifier is a BOOLEAN value, MODIFY sets the value to be `T` (equivalent to using the keyword ADD to add the routine modifier).

For example, both of the following statements alter the **func1** UDR so that it can be executed in parallel in the context of a parallelizable data query statement:

```
ALTER ROUTINE func1 WITH (MODIFY PARALLELIZABLE);
ALTER ROUTINE func1 WITH (ADD PARALLELIZABLE);
```

Example of Altering Routine Modifiers

Suppose you have an external function **func1** that is set to handle NULL values and has a cost per invocation set to `40`. The following ALTER ROUTINE statement adjusts the settings of the function by dropping the ability to handle NULL values, tunes the **func1** by changing the cost per invocation to `20`, and indicates that the function can execute in parallel:

```
ALTER ROUTINE func1(CHAR, INT, BOOLEAN)
   WITH (
      DROP HANDLESNULLS,
      MODIFY PERCALL_COST = 20,
      ADD PARALLELIZABLE
      );
```

Because the name **func1** is not unique to the database, the data type parameters are specified so that the routine signature is unique. If this function had a Specific Name, for example, **raise_sal**, specified when it was created, you could identify the function with the following first line:

```
ALTER SPECIFIC ROUTINE raise_sal;
```

ALTER SECURITY LABEL COMPONENT statement

Use the ALTER SECURITY LABEL COMPONENT statement to add one or more new elements to an existing security label component in the current database. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
ALTER SECURITY LABEL COMPONENT component ADD { ARRAY[ element { BEFORE | AFTER } old_element ] | SET{ element } | TREE( elementUNDERold_element ) }
```

Element	Description	Restrictions	Syntax
component	Component to which *element* is added	Must already exist in the database.	Identifier on page 1231
element	New element of *component*	Must be unique among elements of this *component*, and no longer than 32 bytes. The left (`(`) and right (`)`) parentheses, comma (`,`), and colon (`:`) symbols are not valid characters.	Quoted String on page 1202
old_element	Existing element of *component*	Must be an element of *component*.	Quoted String on page 1202

Usage

Only DBSECADM can issue the ALTER SECURITY LABEL COMPONENT statement, which defines new elements of an existing security label component. The new elements become part of any security policy defined in a CREATE SECURITY POLICY statement that references the specified component.

A security label component consists of a set of no more than 64 *elements* that the CREATE SECURITY LABEL COMPONENT statement defines as string constants. Each string constant can have no more than 32 bytes, and each must be unique among elements of this component. The declaration of each element, which is a valid value that the component can have, defines a category of data sensitivity. By adding new elements to an existing component, the ALTER SECURITY LABEL COMPONENT statement expands the set of values that a component can have within a security policy that includes the component, or within a security label that supports the security policy.

When the ALTER SECURITY LABEL COMPONENT statement executes successfully, Informix® updates the following tables of the system catalog of the current database:

- The **sysseclabelcomponentelements** table, to add new rows for each new elements of the component,
- The **sysseclabelcomponents** table, to show the new cardinality of the security elements that comprise the modified security component.

This statement can define new elements of a security label component, but it cannot modify or drop an existing element. If the security design changes so that different elements are required, DBSECADM can add the new elements, if the total number of elements remains within the size and cardinality limits, and not use any obsolete elements in defining labels that include the component.

Alternatively, DBSECADM can use the DROP SECURITY LABEL COMPONENT statement to drop the component, and then use the CREATE SECURITY LABEL COMPONENT statement to redefine a new component that has only the required elements. You cannot, however, drop a security component if it is part of an existing security policy. See DROP SECURITY statement

on page 485 for information about restrictions on dropping security label components and other security objects of Informix®.

The security label component to which new elements are added must be one of three component types. The ARRAY, SET, or TREE keyword that immediately follows the *component* name must specify the same component type that the CREATE SECURITY LABEL COMPONENT statement specified when the component was originally defined. The syntax for specifying the new list of elements depends on whether the specified component is of type ARRAY, SET, or TREE, which are the three types of security component that Informix® supports.

The ADD ARRAY Clause

A security label component of type ARRAY is an ordered set of no more than 64 elements. The order in which array elements are declared is significant, because it defines a descending order of data sensitivity, with each successive element ranking lower in data sensitivity than the preceding element. The set of label elements of the array and their comma (,) separators must be enclosed between a pair of bracket (`[ ... ]`) symbols. The same new *element* cannot be declared more than once in the same ADD ARRAY clause.

In the ADD ARRAY clause, the BEFORE or AFTER keyword must follow the new element (or a comma-separated list of new elements) to specify the position of the new element within the descending order of data sensitivity. Within restrictions on the size and the number of elements, this syntax enables DBSECADM to insert a new element anywhere in the array, including the highest or the lowest position, or between consecutive existing elements. The ALTER SECURITY LABEL COMPONENT statement fails with an error, however, if the BEFORE or AFTER keyword of the ADD ARRAY clause specifies an array element that was not previously defined, either when the array component was created, or else in a previous ALTER SECURITY LABEL COMPONENT statement that modified the same array component.

If multiple ALTER SECURITY LABEL COMPONENT operations are performed to add elements to the same component of type ARRAY, DBSECADM might not be able to reach the maximum of 64 array elements because of how array elements are encoded. For information on how security elements are encoded, see the HCL Informix® Security Guide.

The following example defines a security label component of type ARRAY called **aquilae** that is an ordered set of five elements, with **imperator** highest in data sensitivity and **asinus** lowest. The subsequent ALTER SECURITY LABEL COMPONENT statement adds two new elements:

- a new element called **legatus** that ranks between **imperator** and **tribunus**
- a new element called **cunctator** that ranks below **asinus** as the new low in data sensitivity.

```
CREATE SECURITY LABEL COMPONENT aquilae
   ARRAY [ "imperator", "tribunus", "centurio", "miles", "asinus" ];

ALTER SECURITY LABEL COMPONENT aquilae
   ADD ARRAY [ "legatus" BEFORE "tribunus","cunctator" AFTER "asinus" ];
```

Successful execution of this ALTER SECURITY LABEL COMPONENT ... ADD ARRAY statement modifies the **aquilae** security label component array, so that the new descending order of component elements becomes this: **imperator, legatus, tribunus, centurio, miles, asinus, cunctator**.

The ADD SET Clause

A security label component of type SET is an unordered set of no more than 64 elements. The order in which the elements of a SET component are declared is not significant. The set of elements of the array and their comma separators must be enclosed between a pair of braces (`{ ... }`) symbols. The same new *element* cannot be declared more than once in the same ADD SET clause.

The following example defines a type SET security label component called **departments** that is an unordered set of three elements, called **Marketing**, **HR**, and **Finance**, which the ALTER SECURITY LABEL COMPONENT statement modifies by adding three new elements called **Training** , **QA**, and **Security**:

```
CREATE SECURITY LABEL COMPONENT departments
   SET { 'Marketing', 'HR', 'Finance' };

ALTER SECURITY LABEL COMPONENT departments
   ADD SET { 'Training', 'QA', 'Security' };
```

Unlike ADD ARRAY or ADD TREE specifications, ADD SET operations of ALTER SECURITY LABEL COMPONENT do not create "greater than" or "less than" data-sensitivity relationships among the new and existing elements of the redefined component, because the elements of a type SET component have no implicit order of data sensitivity.

The ADD TREE Clause

A security label component of type TREE has the logical topology of a simple graph with no loops. Each TREE component has a single root node and no more than 63 additional nodes. Any new elements that the ALTER SECURITY LABEL COMPONENT statement adds to this hierarchy must be inserted below the root node. The string constant for each new node must be followed by the keyword UNDER and by the string constant for some previously declared node. The set of elements of the TREE component, including their UNDER keywords and comma separators, must be enclosed between a pair of parenthesis (`( ... )`) symbols.

The component element specified after the UNDER keyword is called the *parent* of the new element that precedes the same UNDER keyword. The new element is called the *child* of that parent element. The ALTER SECURITY LABEL COMPONENT statement fails with an error, however, if the ADD TREE clause specifies a parent element that is not already defined in the database for this component. The UNDER keyword cannot be followed by an element that is added to the component in the same ADD TREE clause.

The string constant that designates the root node of a TREE component has the highest data sensitivity of all the nodes within the TREE hierarchy. In any subset of successive parent nodes and child nodes in the tree, each non-root element has lower data sensitivity than its parent element or than any ancestor of its parent element, but has higher data sensitivity than any of its child elements or than the descendents of its child elements.

When a user who holds no exemptions attempts to access a data row that is protected by a label that includes a TREE component, a read operation fails if the security label of the user does not include an element that matches one of the TREE elements for the same component of the data row label, or that matches an ancestor of one of those elements. Unless the security policy of the label includes the OVERRIDE clause, a write operation also fails in the same circumstances. If the data row label has multiple TREE components, the user security label must include a matching (or an ancestral) element value for every TREE component of the data row security label.

In the following example, the ALTER SECURITY LABEL COMPONENT statement modifies a tree component called **Oakland** by adding two new nodes to its tree structure that was defined with six nodes by this CREATE SECURITY LABEL COMPONENT statement:

```
CREATE SECURITY LABEL COMPONENT Oakland
TREE ( 'Port' ROOT,
       'Downtown' UNDER 'Port',
       'Airport' UNDER 'Port',
       'Estuary' UNDER 'Airport',
       'Avenues' UNDER 'Downtown',
       'Hills' UNDER 'Avenues');

ALTER SECURITY LABEL COMPONENT Oakland
   ADD TREE ( 'Uptown' UNDER 'Port',
              'Bay' UNDER 'Estuary');
```

Here new **Uptown** node is a child of **Port**, which has the highest data sensitivity because it is the root node. The new **Bay** node is the child of **Estuary**, which is the child of **Airport**, which is the child of **Port**, implying that **Bay** has a lower data sensitivity than these three nodes of the hierarchy. In practice, it is unlikely that any data would be labeled with **Port**, rather than classified at a lower level. The **Port** value might be used for a label granted to a user who is allowed to access all of the data about the **Port**.

If the ALTER SECURITY LABEL COMPONENT statement in this example succeeds, and a subsequently defined data row label specifies **Bay** as its value for the **Oakland** component, a user with no exemption for the security policy who attempts to read the protected row in a query would need either **Port**, **Airport**, **Estuary**, or **Bay** as a user label value to satisfy this component of the data row label. Values of **Uptown** or **Downtown** for this component in the user label are insufficient, because they do not match **Bay** and are not ancestors of **Bay**. For a query to read the protected row, the security label of the user must also include values that satisfy any other components of the row security label, and the user must also hold Select access privilege on the table and at least Connect access privilege on the database that contains the protected row.

The ADD TREE clause cannot interpose a new node between an existing child node and its parent.

ALTER SEQUENCE statement

Use the ALTER SEQUENCE statement to modify the definition of a sequence object. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_125_Connect_42_altseq002) unique_125_Connect_42_altseq002(explicit id unique_125_Connect_42_altseq003) unique_125_Connect_42_altseq003 **ALTER SEQUENCE** [*owner.*] *sequence* [{ | { [] { **NOCYCLE** | **CYCLE** } | [] { **CACHE** *size* | **NOCACHE** } | [] { **NOORDER** | **ORDER** } } | { [] **INCREMENT BY** *step* | [] **RESTART WITH** *restart* | [] { **MAXVALUE** *max* | **NOMAXVALUE** } | [] { **MINVALUE** *min* | **NOMINVALUE** } } }]

Element	Description	Restrictions	Syntax
max	New upper limit on values	Must be integer > **CURRVAL** and *restart*	Literal Number on page 1198

Element	Description	Restrictions	Syntax
min	New lower limit on values	Must be integer < **CURRVAL** and *restart*	Literal Number on page 1198
owner	Owner of *sequence*	Cannot be changed by this statement	Owner name on page 1262
restart	New first value in sequence	Must be integer in the INT8 range	Literal Number on page 1198
sequence	Name of existing sequence	Must exist. Cannot be a synonym.	Identifier on page 1231
size	New number of values to preallocate in memory	Integer > 2 but < cardinality of values in one cycle (= |(*max* - *min*)/*step*|)	Literal Number on page 1198
step	New interval between successive values	Must be a nonzero integer	Literal Number on page 1198

Usage

The ALTER SEQUENCE statement can update the definition of a specified sequence object in the **syssequences** system catalog table.

ALTER SEQUENCE redefines an existing sequence object. It only affects subsequently generated values (and any unused values in the sequence cache). You cannot use the ALTER SEQUENCE statement to rename a sequence nor to change the owner of a sequence.

You must be the owner, or the DBA, or else have the Alter privilege on the sequence to modify its definition. Only elements of the sequence definition that you specify explicitly in the ALTER SEQUENCE statement are modified. An error occurs if you make contradictory changes, such as specifying both MAXVALUE and NOMAXVALUE, or both the CYCLE and NOCYCLE options.

Examples

The examples below are based on the following sequence object and table:

```
CREATE SEQUENCE seq_2
   INCREMENT BY 1 START WITH 1
   MAXVALUE 30 MINVALUE 0
   NOCYCLE CACHE 10 ORDER;

CREATE TABLE tab1 (col1 int, col2 int);
INSERT INTO tab1 VALUES (0, 0);

INSERT INTO tab1 (col1, col2) VALUES (seq_2.NEXTVAL, seq_2.NEXTVAL)


SELECT * FROM tab1;

       col1        col2

          0           0
```

```
          1           1


ALTER SEQUENCE seq_2
   RESTART WITH 5
   INCREMENT by 2
   MAXVALUE 300;

INSERT INTO tab1 (col1, col2) VALUES (seq_2.NEXTVAL, seq_2.NEXTVAL)
INSERT INTO tab1 (col1, col2) VALUES (seq_2.NEXTVAL, seq_2.NEXTVAL)
SELECT * FROM tab1;

        col1        col2

          0           0
          1           1
          5           5
          7           7
```

INCREMENT BY Option

Use the INCREMENT BY option to specify a new interval between successive numbers in a sequence. The interval, or *step* value, can be a positive whole number (for ascending sequences) or a negative whole number (for descending sequences) in the INT8 range. The BY keyword is optional.

RESTART WITH Option

Use the RESTART WITH option to specify a new first number of the sequence. The *restart* value must be an integer within the INT8 range that is greater than or equal to the *min* value (for an ascending sequence) or that is less than or equal to the *max* value (for a descending sequence), if *min* or *max* is specified in the ALTER SEQUENCE statement. The WITH keyword is optional.

When you modify a sequence using the RESTART option, the *restart* value is stored in the **syssequences** system catalog table only until the first use of the sequence object in a **NEXTVAL** expression. After that, the value is reset in the system catalog. Use of the **dbschema** utility can increment sequence objects in the database, creating gaps in the generated numbers that might not be expected in applications that require serialized integers.

MAXVALUE or NOMAXVALUE Option

Use the MAXVALUE option to specify a new upper limit of values in the sequence. The maximum value, or *max*, must be an integer in the INT8 range that is greater than *sequence*.**CURRVAL** and *restart* (or greater than the *origin* in the original CREATE SEQUENCE statement, if *restart* is not specified).

Use the NOMAXVALUE option to replace the current limit with a new default maximum of `2e64` for ascending sequences or `-1` for descending sequences.

MINVALUE or NOMINVALUE Option

Use the MINVALUE option to specify a new lower limit of values in the sequence. The minimum value, or *min*, must be an integer the INT8 range that is less than *sequence***.CURRVAL** and *restart* (or less than the *origin* in the original CREATE SEQUENCE statement, if *restart* is not specified).

Use the NOMINVALUE option to replace the current lower limit with a default of `1` for ascending sequences or `-(2e64)` for descending sequences.

CYCLE or NOCYCLE Option

Use the CYCLE option to continue generating sequence values after the sequence reaches the maximum (ascending) or minimum (descending) limit, to replace the NOCYCLE attribute. After an ascending sequence reaches *max*, it generates the *min* value for the next value. After a descending sequence reaches *min*, it generates the *max* value for the next sequence value.

Use the NOCYCLE option to prevent the sequence from generating more values after reaching the declared limit. Once the sequence reaches the limit, the next reference to *sequence***.NEXTVAL** returns an error message.

CACHE or NOCACHE Option

Use the CACHE option to specify a new number of sequence values that are preallocated in memory for rapid access. The cache size must be a whole number in the INT range that is less than the number of values in a cycle (or less than (|*max* - *min*)/*step*|). The minimum size is `2` preallocated values.

Use NOCACHE to have no values preallocated in memory. (See also the description of SEQ_CACHE_SIZE in CREATE SEQUENCE statement on page 302.)

ORDER or NOORDER Option

These keywords have no effect on the behavior of the sequence. The sequence always issues values to users in the order of their requests, as if the ORDER keyword were always specified. The ORDER and NOORDER keywords are accepted by the ALTER SEQUENCE statement, however, for compatibility with implementations of sequence objects in other dialects of SQL.

ALTER TABLE statement

Use the ALTER TABLE statement to modify the schema of an existing table.

Syntax

(explicit id unique_134_Connect_42_alttab002) unique_134_Connect_42_alttab002(explicit id unique_134_Connect_42_alttab003)
unique_134_Connect_42_alttab003(explicit id unique_134_Connect_42_bastab002) unique_134_Connect_42_bastab002(explicit
id unique_134_Connect_42_bastab016) unique_134_Connect_42_bastab016(explicit id unique_134_Connect_42_bastab003)
unique_134_Connect_42_bastab003(explicit id unique_134_Connect_42_bastab017) unique_134_Connect_42_bastab017(explicit
id unique_134_Connect_42_bastab005) unique_134_Connect_42_bastab005(explicit id unique_134_Connect_42_bastab006)
unique_134_Connect_42_bastab006(explicit id unique_134_Connect_42_bastab007) unique_134_Connect_42_bastab007(explicit
id unique_134_Connect_42_bastab009) unique_134_Connect_42_bastab009(explicit id unique_134_Connect_42_bastab004)
unique_134_Connect_42_bastab004(explicit id unique_134_Connect_42_bastab014) unique_134_Connect_42_bastab014(explicit
id unique_134_Connect_42_bastab008) unique_134_Connect_42_bastab008(explicit id unique_134_Connect_42_bastab010)
unique_134_Connect_42_bastab010(explicit id unique_134_Connect_42_bastab012) unique_134_Connect_42_bastab012(explicit id
unique_134_Connect_42_bastab013) unique_134_Connect_42_bastab013(explicit id unique_134_Connect_42_bastab015) unique_134_Connect_42_bastab015

ALTER TABLE { *table* | *synonym* } { <Basic Table Options> | <Logging TYPE Options> [] | <ADD TYPE Clause> [] | <Statistics Options> [] }

Basic Table Options

" { { | <ADD Column Clause> [] | <ADD AUDIT Clause> [] | <ADD CONSTRAINT Clause> [] | <Add or drop specialized columns> [] | <DROP AUDIT Clause> [] | <DROP CONSTRAINT Clause> [] | <DROP Column Clause> [] | [] <LOCK MODE Clause> [] | <MODIFY Clause> [] | [] <MODIFY EXTENT SIZE Clause> [] | [] <MODIFY NEXT SIZE Clause> [] | <PUT Clause> [] | <SECURITY POLICY Clause> [] } } "

Element	Description	Restrictions	Syntax
synonym	Synonym for the table to be altered	Synonym and its table must exist; **USETABLENAME** must not be set	Identifier on page 1231
table	Name of table to be altered	Must exist in the current database	Identifier on page 1231

Usage

You can use the Basic Table Options segment to modify the schema of a table by adding, modifying, or dropping columns and constraints, or changing the extent size or locking granularity of a table. The database server performs alterations in the order that you specify. If any of the actions fails, the entire operation is canceled. You can associate an existing table with a named ROW type, or specify a new storage space to store large-object data. You can add or drop shadow columns to support secondary-server update operations of the USELASTCOMMITTED feature, or add or drop a **rowid** column. However, a single ALTER TABLE statement cannot specify these options with most other alterations to the schema of the table.

To use ALTER TABLE, your discretionary access privileges must meet at least one of the following conditions:

- You must have DBA privilege on the database that contains the table.
- You must own the table.
- You must have the Alter privilege on the specified table and the Resource privilege on the database where the table resides.
- To add a referential constraint, you must have the DBA or References privilege on either the referenced columns or the referenced table.
- To drop a constraint, you must have the DBA privilege or be the owner of the constraint. If you are the owner of the constraint but not the owner of the table, you must have Alter privilege on the specified table. You do not need the References privilege to drop a constraint.

If you run a slow alter on a table that contains a compressed partition, then the corresponding new partition is compressed. The number of compressed rows in the new partition might differ from the number of compressed rows in the original partition. The difference is caused by the timing of operations to rewrite the rows in the partition and to build the compression dictionary. If the resulting new partition has fewer compressed rows, you can recompress the partition, and optionally repack and shrink it.

The ALTER TABLE statement cannot add a fragmentation strategy to a nonfragmented table, nor modify the storage distribution strategy of a fragmented table. To modify the distributed storage strategy of a table, you must use the ALTER FRAGMENT statement, rather than the ALTER TABLE statement. For information on adding, modifying, or dropping the storage distribution strategy of a table, see the ALTER FRAGMENT statement on page 23.

Altering a table on which a view depends might invalidate the view.



Warning: The clauses available with this statement have varying performance implications. Before you undertake alter operations, check information in the Altering a table definition on page section in your to review effects and strategies.

The table whose name or synonym follows the ALTER TABLE keywords must be a permanent table in the current database. It is subject to the following restrictions:

- It cannot be a temporary table.
- It cannot be a table in a database that is not the current database.
- It cannot be a table object that the CREATE EXTERNAL TABLE statement defined.
- It cannot be a violations table or a diagnostics table.
- If the **USETABLENAME** environment variable is set, you cannot specify a *synonym* for the table in the ALTER TABLE statement.

In addition, you cannot use the ALTER TABLE statement for the following operations:

- Add, drop, or modify a column in a table that has an associated violation table or diagnostics table.
- Define a referential constraint or a unique constraint on a RAW table.
- Define an index on a column or on a set of columns that would conflict with the Restrictions on columns as index keys on page 244.

Add or drop specialized columns

You can add or drop specialized columns when you alter a table. Most specialized columns are shadow columns that support Enterprise Replication or high availability.

This syntax fragment is part of the ALTER TABLE statement on page 95.

Adding or dropping specialized columns

```
" { ADD | DROP } { CRCOLS | ERKEY | REPLCHECK | ROWIDS | VERCOLS } "
```

Usage

The following keywords control one or more specialized columns.

CRCOLS

The ADD CRCOLS keywords create shadow columns, **cdrserver** and **cdrtime**, that uses for conflict resolution. If Enterprise Replication is active while you are altering the table with the ADD CRCOLS clause, you must first put the table in alter mode with the cdr alter command. Altering a table to add the CRCOLS shadow columns can be a slow alter operation, if any of the table columns have data types that require a slow alter. Slow alter operations require disk space at least twice the size of the original table plus log space. For information on the performance implications of ALTER TABLE statements, see Altering a table definition on page .

Use the DROP CRCOLS keywords to drop the **cdrserver** and **cdrtime** shadow columns. You must stop replication before you drop the **cdrserver** and **cdrtime** shadow columns.

For more information, refer to Using the WITH CRCOLS Option on page 346.

ERKEY

The ADD ERKEY keywords create shadow columns, **ifx_erkey_1**, **ifx_erkey_2**, and **ifx_erkey_3**, that uses in place of a primary key. If Enterprise Replication is active while you are altering the table with the ADD ERKEY clause, you must first put the table in alter mode with the cdr alter command.Altering a table to add the ERKEY shadow columns is a slow alter operation.

Use the DROP ERKEY keywords to drop the **ifx_erkey_1**, **ifx_erkey_2**, and **ifx_erkey_3** shadow columns.

For more information, see Using the WITH ERKEY Keywords on page 347.

REPLCHECK

The ADD REPLCHECK keywords create the shadow column, **ifx_replcheck**, that you can create an index on, along with your primary key, to speed the processing of consistency checking with . If Enterprise Replication is active while you are altering the table with the ADD REPLCHECK clause, you must first put the table in alter mode with the cdr alter command.Altering a table to add the **ifx_replcheck** shadow column is a slow alter operation, which requires disk space at least twice the size of the original table plus log space.

Use the DROP REPLCHECK keywords to drop the **ifx_replcheck** shadow column.

For more information, see Using the WITH REPLCHECK Keywords on page 347.

ROWIDS

Important: This is a deprecated feature. The query optimizer might not use an index scan when explicit **rowid** shadow columns are defined on fragmented tables. When you create new applications, use primary keys as a method of row identification instead of using **rowid** values.

Use the ADD ROWIDS keywords to add a column called **rowid** to a fragmented table. (Fragmented tables do not contain the hidden **rowid** column by default.) When you add a **rowid** column, the database server assigns a unique number to each row that remains stable for the life of the row. The database server creates an index that it uses to find the physical location of the row. After you add the **rowid** column, each row of the table contains an additional 4 bytes to store the **rowid** value. Use the ADD ROWIDS clause only on fragmented tables. In nonfragmented tables, the **rowid** column remains unchanged.

The DROP ROWIDS keywords can drop a **rowid** column that you added (with either the ALTER TABLE or ALTER FRAGMENT statement) to a fragmented table.

VERCOLS

The ADD VERCOLS keywords create shadow columns, **ifx_insert_checksum** and **ifx_row_version**, that are used to support secondary server updates. Altering a table to add row versioning columns is a fast alter operation.

Use the DROP VERCOLS keywords to drop the **ifx_insert_checksum** and **ifx_row_version** shadow columns.

For more information, see Using the WITH VERCOLS Option on page 348 and Row versioning on page .

Example

Example: ADD CRCOLS

In the following example, the **cdrserver** and **cdrtime** shadow columns are added to the **customer** table:

```
ALTER TABLE customer ADD CRCOLS;
```

Example

Example: ADD ERKEY

The following example adds the **ifx_erkey_1**, **ifx_erkey_2**, and **ifx_erkey_3** columns to the **customer** table:

```
ALTER TABLE customer ADD ERKEY;
```

Example

Example: ADD REPLCHECK and DROP REPLCHECK

In the following example, the **ifx_replcheck** shadow column is added to the **customer** table:

```
ALTER TABLE customer ADD REPLCHECK;
```

The following example drops the **ifx_replcheck** column from the **customer** table:

```
ALTER TABLE customer DROP REPLCHECK;
```

Example

Example: DROP ROWIDS

The following example drops the **rowid** column from the **frag1** table:

```
ALTER TABLE frag1 DROP RWIDS;
```

Example

Example: ADD VERCOLS

In the following example, the **ifx_insert_checksum** and **ifx_row_version**, shadow columns are added to the **customer** table:

```
ALTER TABLE customer ADD VERCOLS;
```

ADD Column Clause

Use the ADD Column clause to add a column to a table, and to define constraints on the new column. This clause can also associate a security policy with a table that has no security policy, or can specify a security label for the new column, if the table already has a security policy.

This syntax fragment is part of the ALTER TABLE statement on page 95.

(explicit id unique_137_Connect_42_addc002c) unique_137_Connect_42_addc002c(explicit id unique_137_Connect_42_addc003c)
unique_137_Connect_42_addc003c(explicit id unique_137_Connect_42_addc004c) unique_137_Connect_42_addc004c(explicit
id unique_137_Connect_42_addc005c) unique_137_Connect_42_addc005c(explicit id unique_137_Connect_42_addc006c)
unique_137_Connect_42_addc006c(explicit id unique_137_Connect_42_addc007c) unique_137_Connect_42_addc007c

ADD Column Clause

" **ADD** "

" { (<New Column>) <New Column> } "

New Column

" *new_column* "

" <Data Type> [] "

" [{ | [[] <DEFAULT Clause> []] | [[] <Single-Column Constraint Format> []] }] "

" [**BEFORE** *column* []] "

" [<Add Column SECURED WITH label clause> []] "

Element	Description	Restrictions	Syntax
column	Name of column before which *new_column* is to be placed	Must already exist in the table	Identifier on page 1231
new_column	Name of column that you are adding	You cannot add a serial column if the table contains data	Identifier on page 1231

The following restrictions apply to the ADD clause:

- You cannot add a serial column to a table that contains data.
- You cannot add columns beyond the maximum row size of 32,767 bytes.

Restrictions on adding columns of type IDSSECURITYLABEL

The following restrictions affect the use of the ADD Column clause to add a column of the IDSSECURITYLABEL data type to support a label-based access control (LBAC) security policy:

- If the table has no security policy, a user who holds the DBSECADM role must also include the `ADD SECURITY POLICY` keywords to specify an LBAC security policy, as in this example:

```
CREATE TABLE tA
   (Col_1 BIGINT
    Col_2 INTERVAL(YEAR TO MONTH);
    Col_3 VARCHAR (255));
 . . .
ALTER TABLE tA
  ADD (Col_4 IDSSECURITYLABEL DEFAULT 'label4'), --to add this column
  ADD SECURITY POLICY Watchdog;   --a security policy is also required
                               -- and 'label4' must be a Watchdog label
```

- Only a user who holds the DBSECADM role can add a column of type IDSSECURITYLABEL.
- A table can have at most one column of type IDSSECURITYLABEL.
- The IDSSECURITYLABEL column cannot have `SECURED WITH` column protection.
- The IDSSECURITYLABEL column has an implicit NOT NULL constraint by default. If no *label* name for the default security label is specified in the DEFAULT clause, the default value for this column is the security label for write access that is held by the current user.
- The IDSSECURITYLABEL column cannot have any explicit single-column constraints, and it cannot be part of multiple-column referential or check constraints.

For the ALTER TABLE syntax to add or drop row-level LBAC protection for the table, see SECURITY POLICY Clause on page 154.

For the ALTER TABLE syntax to add column-level protection for a new column in a table already protected by an LBAC security policy, see Add column SECURED WITH label clause on page 113.

For the ALTER TABLE syntax to add or drop column-level LBAC protection for an existing column, see Modify Column Security clause on page 144.

Logical Character Support in Character Columns

For new columns that you declare as built-in character data types, explicit or default size specifications are interpreted in units of bytes, unless the SQL_LOGICAL_CHAR configuration parameter has enabled logical character semantics for the current database. This feature is designed to reduce the risk of truncating data strings in locales that support a multibyte code set, such as **UTF-8**. Enabling this feature causes the SQL parser to interpret the declared column size as units of logical characters, rather than as bytes, and multiplies the declared storage size allocated for the new character column by a positive integer value, based on the SQL_LOGICAL_CHAR setting.

- If the value of this setting is `OFF` or `1`, the SQL_LOGICAL_CHAR configuration parameter has no effect.
- If the value of this setting is `ON`, rather than a digit, the expansion factor is the number of bytes that are required to store the largest logical character in the code set of the database. (The `ON` setting is equivalent to `4`, which is the largest valid digit.)

The value of this expansion factor is an attribute of the database, and is based on the SQL_LOGICAL_CHAR setting when the database was created, rather than when the ALTER TABLE statement is issued, if the two settings are not identical.

For columns that you declare as VARCHAR or NCHAR data types when this feature is enabled, only the maximum size specification is expanded by this feature. The reserved size is the number of bytes specified by the explicit or default *reserved* value in the data type declaration, because the minimum size of a logical character is 1 byte.

Size specifications for character columns of user-defined types (UDTs) are always interpreted as bytes, and are not affected by this feature. Columns that store strings as large objects, such as CLOB and TEXT, are similarly unaffected.

For more information about the SQL_LOGICAL_CHAR configuration parameter, see your *HCL® Informix® Administrator's Reference*. For additional information about multibyte locales and logical characters, see the *HCL® Informix® GLS User's Guide*.

BEFORE Clause

The optional BEFORE clause determines the ordinal positions of the new columns within the schema of the table by specifying the name of an existing column before which the ALTER TABLE ADD statement inserts the new columns.

About this task

In the following example, the BEFORE option directs the database server to add the **item_weight** column before the **total_price** column:

```
ALTER TABLE items
   ADD (item_weight DECIMAL(6,2) NOT NULL BEFORE total_price);
```

By default, if you do not include the BEFORE clause, the database server appends the new columns after the last column in the current schema of the table, in their lexical order within the ADD clause.

DEFAULT clause of ALTER TABLE

Use the DEFAULT clause of the ALTER TABLE statement to specify value that the database server should insert in a column in DML operations that specify no explicit value for the column.

This syntax fragment is part of the ADD Column Clause on page 100 and the MODIFY Clause on page 139.

(explicit id unique_156_Connect_42_defc001) unique_156_Connect_42_defc001(explicit id unique_156_Connect_42_defc002) unique_156_Connect_42_defc002

DEFAULT Clause

" DEFAULT "

" { *label* | *literal* | { NULL | { USER | CURRENT_USER } } | [] { CURRENT | SYSDATE } [<DATETIME Field Qualifier> []] | TODAY | { SITENAME | DBSERVERNAME } } "

Element	Description	Restrictions	Syntax
label	Name of a security label	Must exist and must belong to the security policy that protects the table. The column must be of type IDSSECURITYLABEL.	Identifier on page 1231

Element	Description	Restrictions	Syntax
literal	Literal default value for the column	Must be appropriate for the data type of the column. See Using a Literal as a Default Value on page 325.	Expression on page 1003

Usage

If the table that you are altering already has rows in it when you use the ALTER TABLE ADD statement to add a column that contains a default value, the default values are applied to all existing rows and rows inserted after the ALTER TABLE ADD statement that added the new column.

Similarly, when the ALTER TABLE MODIFY statement uses the DEFAULT clause to define a new default value for a column that had no default or that had a different default, the rows that existed before the column was modified are unchanged, unless you update those rows to insert some NULL or non-NULL value. New rows that you insert will have the default value that the DEFAULT clause of the ALTER TABLE MODIFY statement specified, unless you insert some other value into the new column.

You cannot specify a default value for columns of type SERIAL, SERIAL8, or BIGSERIAL. For information about using the ALTER TABLE MODIFY statement to set the next value of a serial column to an arbitrary value higher than the current maximum, see Altering the Next Serial Value on page 141.

For columns of DISTINCT or OPAQUE data types, you cannot specify as the default value a constant expression (such as CURRENT, SYSDATE DBSERVERNAME, SITENAME, TODAY, USER, or CURRENT_USER) that behaves like a variant function.

The DEFAULT NULL keywords

If you use the ALTER TABLE ADD statement to add a new column with an explicit default value, or if you use the ALTER TABLE MODIFY statement to change the data type or the default value of an existing column, the following restrictions apply to `NULL` as a default value:

- For columns of large-object data types like BYTE, TEXT, BLOB, or CLOB, or for columns of key-value pair (KVP) data types like BSON or JSON, the only valid default value is `NULL`.
- If you specify `NULL` as the default value for a column, you cannot specify a NOT NULL constraint as part of the column definition. (For details of NOT NULL constraints, see Using the NOT NULL Constraint on page 328.)
- NULL is not a valid default value for a column that is part of a primary key.
- Serial columns cannot have a default value, including `DEFAULT NULL`.
- If a column was created or altered with no DEFAULT clause, and with no implicit or explicit NOT NULL constraint, its implicit default value is `NULL`.

Examples of default column values

The following statement adds column **item_velocity** to the **items** table, with `NULL` as its implicit default value:

```
ALTER TABLE items
   ADD (item_velocity DECIMAL(6,3) BEFORE total_price);
```

The following statement modifies the default value of column **item_velocity** by replacing `NULL` with an explicit default value:

```
ALTER TABLE items
   MODIFY (item_velocity DECIMAL(6,3) DEFAULT 299792.458);
```

Each existing row in the **items** table has the default value of NULL for the **item_velocity** column. Any new rows have the default value of `299792.458`.

The next example adds a new column of data type CHAR(12) to the **items** table, where the new **item_color** column has an explicit default value of **`translucent`**:

```
ALTER TABLE items
   ADD (item_color CHAR(12) DEFAULT "translucent"
   BEFORE item_velocity);
```

Suppose that table **tabB** has the following schema:

```
CREATE TABLE tabB
  (
    id VARCHAR(128) NOT NULL,
    data DATE DEFAULT TODAY,
    modcount BIGINT,
    flags INTEGER DEFAULT 12,
  );
```

The following statement modifies **tabB** by changing the data type and the default value of the **data** column from type DATE with a default of **TODAY** to a BSON type with a default of NULL:

```
ALTER TABLE tabB
   MODIFY ( data "informix".BSON DEFAULT NULL);
```

The original default value of **TODAY** for the **data** column is not valid for a key-value pair data type like BSON.

For more information about the options of the DEFAULT clause, refer to the DEFAULT clause of CREATE TABLE on page 324 topic of the CREATE TABLE statement.

DEFAULT security labels

When DBSECADM adds a IDSSECURITYLABEL column to a table that is protected by a security policy, the DEFAULT *label* specification is required unless the table is empty. If the table is not empty, the specified *label* is inserted into the existing rows of the table

If the DEFAULT value is the name of a security label, an error is issued in the following cases:

- if the *label* is the default value for a column whose data type is not IDSSECURITYLABEL,
- or if the table has no security policy,
- or if the security policy of the *label* is not the security policy of the table.

The same ALTER TABLE statement that adds the IDSSECURITYLABEL column can add a security policy.

To define a specific *label* as the default value of an IDSSECURITYLABEL column, specify the *label* name without the *policy* qualifier, rather than as *policy*.*label*. The current security policy of the table is the only valid policy for any security label that protects data in the table.

Examples of security labels as default values

The ALTER TABLE statement in the following example adds security policy **MegaCorp** to table **T1** and specifies column-level protection for the table by declaring a new column **D** of type IDSSECURITYLABEL, whose default value is a security label called **mylabel**:

```
ALTER TABLE T1
   ADD (D IDSSECURITYLABEL DEFAULT mylabel1)
   ADD SECURITY POLICY MegaCorp;
```

Because no BEFORE clause is included, column **D** is last among the columns in the schema of table **T1**. This statement fails if any of the database objects that it references (except new column **D**) does not already exist in the database, or if the table already has a different security policy.

To replace the security policy of a table, you must hold the DBSECADM role. You must first use the ALTER TABLE DROP SECURITY POLICY statement to drop the current security policy and any of its labels from the table. Then you must add the new security policy, and at least one of its labels, as in the following example:

```
ALTER TABLE T1
   DROP SECURITY POLICY MegaCorp;
ALTER TABLE T1
   ADD (D IDSSECURITYLABEL DEFAULT myNewLabel1)
   ADD SECURITY POLICY Watchdog;
```

In these ALTER TABLE statements that reference table **T1**,

- The DROP SECURITY POLICY clause of the first statement removes table **T1** from the protection of the **MegaCorp** security policy,
- and automatically drops from the schema of table **T1** any IDSSECURITYLABEL column that stores a label of the **MegaCorp** security policy. This has no effect, however, on other tables in the database that are protected by the **MegaCorp** security policy.
- The `ADD (D IDSSECURITYLABEL DEFAULT myNewLabel1)` clause protects table **T1** with the **myNewLabel1** security label,
- and the `ADD SECURITY POLICY Watchdog` clause replaces **MegaCorp** with **Watchdog** as the new security policy of table **T1**.

The second ALTER TABLE example fails unless the **myNewLabel1** security label is a label of the **Watchdog** security policy.

For more information about using the ALTER TABLE statement to add, modify, or drop an association between a table and a label-based security policy, see Modify Column Security clause on page 144 and SECURITY POLICY Clause on page 154. See also the DDL statements for creating label-based security objects, including CREATE SECURITY POLICY statement on page 298, CREATE SECURITY LABEL statement on page 293, and CREATE SECURITY LABEL COMPONENT statement on page 294.

Single-Column Constraint Format

Use the Single-Column Constraint Format to associate one or more constraints with a single column when you alter a table.

This syntax fragment is part of the ADD Column Clause on page 100 and the MODIFY Clause on page 139.

(explicit id unique_157_Connect_42_singc001) unique_157_Connect_42_singc001(explicit id unique_157_Connect_42_singc002) unique_157_Connect_42_singc002(explicit id unique_157_Connect_42_singc003) unique_157_Connect_42_singc003(explicit id unique_157_Connect_42_singc004) unique_157_Connect_42_singc004

Single-Column Constraint Format

```
" { { NULL | NOT NULL } | { UNIQUE | DISTINCT } | PRIMARY KEY | <REFERENCES Clause> | <CHECK Clause> } [
<Constraint Definition> ] "
```

You cannot specify a primary-key or unique constraint on a new column if the table contains data. In the case of a unique constraint, however, the table can contain a *single* row of data. When you want to add a column with a primary-key constraint, the table must be empty when you issue the ALTER TABLE statement.

The following rules apply when you place primary-key or unique constraints on existing columns:

- When you place a primary-key or unique constraint on a column or on a set of columns, the database server creates an internal B-tree index on the constrained column or set of columns, and automatically calculates column statistics, equivalent to distributions created by the UPDATE STATISTICS statement in HIGH mode, unless a user-created index was already defined on the same column or set of columns.
- When you place a primary-key or unique constraint on a column or set of columns, and a unique index already exists on that column or set of columns, the constraint shares that index. If the existing index allows duplicates, however, the database server returns an error. You must then drop the existing index before you can add the constraint.
- When you place a primary-key constraint or a unique constraint on a column or on a set of columns on which a referential constraint already exists, the existing index that enforces the constraint is upgraded to UNIQUE (if possible), and the index is shared.

You cannot place a unique constraint on a BYTE or TEXT column, nor can you place referential constraints on columns of these data types. A check constraint on a BYTE or TEXT column can check only for IS NULL, IS NOT NULL, or LENGTH.

The statement fails with an error if you specify both a NOT NULL constraint and a NULL constraint on the same column. You cannot define a NULL constraint on a column whose data type is LIST, MULTISET, SET, or IDSSECURITYLABEL.

The IDSSECURITYLABEL column has an implicit NOT NULL constraint, but it cannot have explicit single-column constraints nor be part of multiple-column referential constraints or check constraints. If the constraint is on a column that stores encrypted data, Informix® cannot enforce the constraint.

Important:

You cannot use the Single-Column Constraint Format to add a new column with a foreign-key constraint in ENABLED NOVALIDATE or FILTERING WITH ERROR NOVALIDATE or FILTERING WITHOUT ERROR NOVALIDATE constraint mode. For the ALTER TABLE statement to create a new foreign-key constraint with the `NOVALIDATE` keyword bypassing

violations-checking during the ALTER TABLE operation, you must use the ALTER TABLE ADD CONSTRAINT syntax with the Multiple-Column Constraint Format.

Using NOT NULL Constraints with ADD COLUMN

If a table contains data, when you add a column with a NOT NULL constraint you must also include a DEFAULT clause.

If the table is empty, however, you can add a column and apply only the NOT NULL constraint. The following statement is valid whether or not the table contains data:

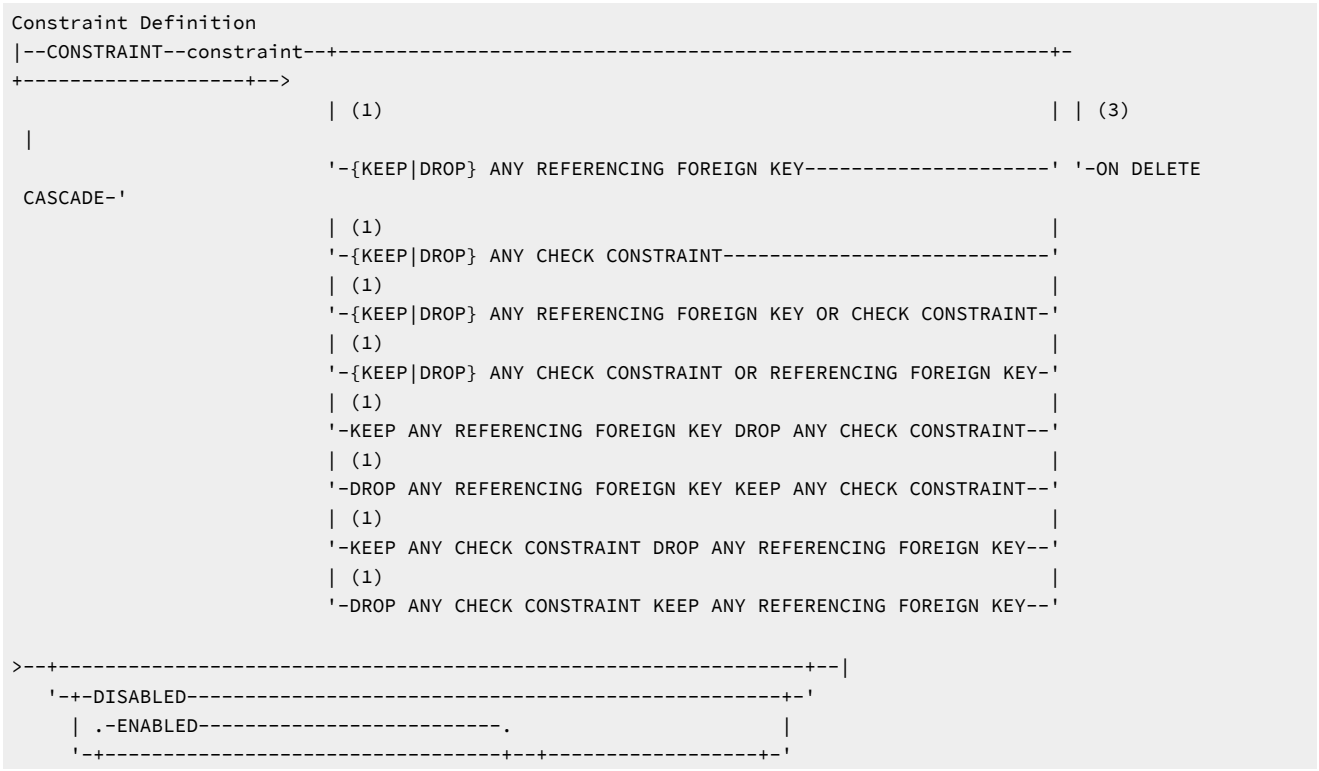
```
ALTER TABLE items
   ADD (item_weight DECIMAL(6,2)
   DEFAULT 2.0 NOT NULL
      BEFORE total_price);
```

Constraint Definition

Use the Constraint Definition segment of the ALTER TABLE statement to declare the name of a constraint, and to set its mode to DISABLED or ENABLED, or for tables that have violations tables, two FILTERING modes.

For enabled or filtering foreign-key or check constraints that the ALTER TABLE ADD CONSTRAINT statements can create, the NOVALIDATE mode can prevent the database server from checking every row of the table for compliance with the enabled constraint while the ALTER TABLE statement is creating the constraint.

Both the Single-Column Constraint Format on page 106 format and the Multiple-Column Constraint Format on page 117 support the following syntax for defining constraints:



```
      |                             (2) |  |  (3) (4)           |
      '-FILTERING--+-WITHOUT ERROR-+-----'  '-------NOVALIDATE-'
                   '-WITH ERROR----'
```

1. Valid for PRIMARY KEY constraints only
2. See Filtering Modes
3. Valid for FOREIGN KEY constraints only
4. Valid for CHECK constraints only

Element	Description	Restrictions	Syntax
constraint	Name declared here for the constraint	Must be unique among the names of indexes and constraints in database	Identifier on page 1231

Usage

If the ALTER TABLE statement includes the Single-Column Constraint format or the Multiple-Column Constraint format, but the Constraint Definition is empty, the database server creates and enables whatever type of constraint the Single-Column Constraint or Multiple-Column Constraint format specified, assigns to the constraint a system-generated identifier and a default object state, and registers these attributes in the **sysconstraints** and **sysobjstate** system catalog tables.

If you specify no mode for the constraint, the constraint is enabled by default.

The optional `ON DELETE CASCADE` keywords can precede or follow the declaration of the constraint name. For referential constraints, the `ON DELETE CASCADE` keywords instructs the database server to delete foreign-key rows from the child tables when it deletes rows with the corresponding primary key from the parent table. For more information on the effects of these keywords on DELETE operations, see Using the ON DELETE CASCADE Option on page 111.

While creating and enabling foreign-key or check constraints that the ALTER TABLE ADD CONSTRAINT statement defines, the `NOVALIDATE` keyword prevents the database server from checking every row of the table for compliance with the constraint while the ALTER TABLE statement is running. For more information on the restrictions and effects of this keyword for foreign-key constraints, see Creating foreign-key constraints in NOVALIDATE modes on page 123.

Just as in the CREATE TABLE statement, you cannot define unique constraints, primary-key constraints, or referential constraints on a BYTE or TEXT column. In addition, the table cannot be a RAW table.

For more information about constraint-mode options, see Choosing a Constraint-Mode Option on page 337.

REFERENCES Clause

This syntax fragment is part of the Single-Column Constraint Format on page 106.

(explicit id unique_171_Connect_42_refc002) unique_171_Connect_42_refc002

REFERENCES Clause

" REFERENCES *table* "

" [(*column*)] "

" [[] ON DELETE CASCADE] "

Element	Description	Restrictions	Syntax
column	Referenced column in the referenced table	See Restrictions on Referential Constraints on page 110.	Identifier on page 1231
table	The referenced table	The referenced and the referencing tables must reside in the same database	Identifier on page 1231

The REFERENCES clause allows you to place a foreign-key constraint on one or more columns. The referenced column can be in the same table as the referencing column, or in a different table in the same database.

For example, the following example declares a disabled foreign-key constraint called **detail_fork1** on the **detail** table.

```
CREATE TABLE master (col_one INT, col_two CHAR);
CREATE TABLE detail (col_one INT, col_twain CHAR);
. . .
ALTER TABLE detail ADD CONSTRAINT (FOREIGN KEY ( col_one )
   REFERENCES master( col_one ) CONSTRAINT detail_fork1 DISABLED);
```

In DISABLED mode, **detail_fork1** has no effect, but if it were subsequently enabled, it would restrict INSERT and UPDATE operations on the **detail** table to rows whose column **col_one** values matched existing values in column **col_one** of the referenced **master** table. In the ALTER TABLE statement above, the parentheses (FOREIGN KEY . . . DISABLED) that delimit the new referential constraint definition are optional.

If the referenced table is different from the referencing table, the default *column* is the primary-key column. If the referenced table is the same as the referencing table, there is no default.

The optional ON DELETE CASCADE keywords can either be specified the last keywords in the REFERENCES clause, or they can follow the declaration of the constraint name in the Constraint definition.

For more information on the effects of these keywords in DELETE operations, see Using the ON DELETE CASCADE Option on page 111.

Restrictions on Referential Constraints

You must have the REFERENCES privilege to create a referential constraint.

The following restrictions apply to the *column* that is specified (the referenced column) in the REFERENCES clause:

- The referenced and referencing tables must be in the same database.
- The referenced column (or set of columns) must have a unique or primary-key constraint.
- The referencing and referenced columns must be the same data type.

  The only exceptions are that a referencing column must be an integer data type if the referenced column is a serial data type:
  - For BIGSERIAL referenced columns, use BIGINT referencing columns.
  - For SERIAL referenced columns, use INT referencing columns.
  - For SERIAL8 referenced columns, use INT8 referencing columns.
- You cannot place a referential constraint on a BYTE or TEXT column.
- You cannot place a constraint on any column of a RAW table.
- Constraints uses the collation in effect at their time of creation.
- A column-level REFERENCES clause can include only a single column name.
- Maximum number of columns in a table-level REFERENCES clause is 16.
- The total length of the columns in a table-level REFERENCES clause cannot exceed 390 bytes.

Default Column for the References Clause

If the referenced table is different from the referencing table, you do not need to specify the referenced column; the default column is the primary-key column (or columns) of the referenced table. If the referenced table is the same as the referencing table, you must specify the referenced column.

The following example creates a new column in the **cust_calls** table, **ref_order**. The **ref_order** column is a foreign key that references the **order_num** column in the **orders** table.

```
ALTER TABLE cust_calls
   ADD ref_order INTEGER
   REFERENCES orders (order_num)
   BEFORE user_id;
```

When you place a referential constraint on a column or set of columns, and a duplicate or unique index already exists on that column or set of columns, the index is shared.

Using the ON DELETE CASCADE Option

Use the ON DELETE CASCADE option if you want rows deleted from the child table when the DELETE or MERGE statement removes corresponding rows from the parent table.

Here the *parent table* is the table specified in the REFERENCING clause of the definition of an enabled foreign key constraint, and the *child table* is the table on which the enabled foreign key constraint is defined. If you do not specify cascading deletes, the default behavior of the database server prevents DELETE and MERGE statements from deleting data in a table that another tables references within a primary-key foreign-key relationship.

If you specify this option, when you delete a row in the parent table, the database server also deletes any rows associated with that row (foreign keys) in a child table. The advantage of the ON DELETE CASCADE option is that it allows you to reduce the quantity of SQL statements needed to perform delete actions.

For example, in the **stores_demo** database, the **stock** table contains the **stock_num** column as a primary key. The **catalog** table refers to the **stock_num** column as a foreign key. The following ALTER TABLE statements drop an existing foreign-key constraint (without cascading delete) and add a new constraint that specifies cascading deletes:

```
ALTER TABLE catalog DROP CONSTRAINT aa;

ALTER TABLE catalog ADD CONSTRAINT
   (FOREIGN KEY (stock_num, manu_code) REFERENCES stock
   ON DELETE CASCADE CONSTRAINT ab);
```

With cascading deletes specified on the child table, in addition to deleting a stock item from the **stock** table, the delete cascades to the catalog table that is associated with the **stock_num** foreign key. This cascading delete works only if the **stock_num** that you are deleting was not ordered; otherwise, the constraint from the **items** table would disallow the cascading delete. For more information, see Restrictions on DELETE When Tables Have Cascading Deletes on page 454.

If a table has a trigger with a DELETE trigger event, you cannot define a cascading-delete referential constraint on that table. You receive an error when you attempt to add a referential constraint that specifies ON DELETE CASCADE to a table that has a delete trigger.

The TRUNCATE statement cannot result in cascading deletes from a child table. The target table of the TRUNCATE statement cannot be referenced in the definition of an enabled foreign-key constraint on another table (unless that child table has no rows).

For information about syntax restrictions and locking implications when you delete rows from tables that have cascading deletes, see Considerations When Tables Have Cascading Deletes on page 454.

Locks Held During Creation of a Referential Constraint

When you create a referential constraint, the database server places an exclusive lock on the referenced table. The lock is released after you finish with the ALTER TABLE statement or at the end of a transaction (if you are altering the table in a database that uses transaction logging).

CHECK Clause

A check constraint designates a condition that must be met before data can be inserted into a column.

This syntax fragment is part of the Single-Column Constraint Format on page 106.

(explicit id unique_172_Connect_42_chec001) unique_172_Connect_42_chec001

CHECK Clause

" **CHECK** "

" (<Condition>) "

During an insert or update, if a row returns *false* for any check constraint defined on a table, the database server returns an error. No error is returned, however, if a row returns NULL for a check constraint. In some cases, you might want to use both a check constraint and a NOT NULL constraint.

Check constraints are defined using *search conditions*. The search condition cannot contain user-defined routines, subqueries, aggregates, host variables, or rowids. In addition, the condition cannot contain the variant built-in functions CURRENT, SYSDATE, USER, CURRENT_USER, SITENAME, DBSERVERNAME, or TODAY.

The check constraint cannot include columns in different tables. When you are using the ADD or MODIFY clause, the check constraint cannot depend upon values in other columns of the same table.

The next example adds a new **unit_price** column to the **items** table and includes a check constraint to ensure that the entered value is greater than `0`:

```
ALTER TABLE items
   ADD (unit_price MONEY (6,2) CHECK (unit_price > 0));
```

To create a constraint that checks values in more than one column, use the ADD CONSTRAINT clause. The following example builds a constraint on the column that was added in the previous example. The check constraint now spans two columns in the table.

```
ALTER TABLE items ADD CONSTRAINT CHECK (unit_price < total_price);
```

When you create or enable a check constraint, you can speed up the statement by including the NOVALIDATE keyword to skip the checking of existing rows for violations. The check constraint is enabled when the statement completes.

Add column SECURED WITH label clause

The SECURED WITH Label clause associates the new column that the ALTER TABLE ADD statement defines with a label-based access control (LBAC) security label. This clause is valid only for a table that is protected by an LBAC security policy, and for a security label of the same policy.

This syntax fragment is part of the ADD Column Clause on page 100.

COLUMN SECURED WITH Label clause

" [COLUMN] SECURED WITH*label* "

Element	Description	Restrictions	Syntax
label	Name of an LBAC security label	Must exist and must belong to the same LBAC security policy that protects the table.	Identifier on page 1231

Usage

In a database that supports label-based access control, only tables that have a column of type IDSSECURITYLABEL can be protected by an LBAC security policy. When the ALTER TABLE ADD statement adds a new column to a protected table, the COLUMN SECURED WITH Label clause can associate the column with a security label of the same security policy.

- In a protected table, any data in a column that the SECURED WITH label clause associates with an LBAC security label is protected with *column-level granularity* for those columns. Only users whose security credentials satisfy the label of the column can access data in the column.
- In the same table, the data in other columns not referenced by any SECURED WITH label clause are protected with *row-level granularity* by the LBAC label in the column of type IDSSECURITYLABEL. Only users whose security credentials satisfy that label can access data in any row.

Only users who hold the DBSECADM role can include this clause in the ALTER TABLE statement.

The COLUMN SECURED WITH Label clause is an extension to the ISO/ANSI standard for the SQL language.

The COLUMN keyword is optional, and has no effect on the result of the ALTER TABLE SECURED WITH operation, but it might make your code easier for human readers to understand.

The ALTER TABLE statement cannot add a security label to table objects in any of the following categories:

- virtual-table interface (VTI) tables or virtual-index interface (VII) tables
- user-defined or system-defined temporary (TEMP) tables
- tables of named or unnamed ROW types
- parent tables or child tables within a typed-table hierarchy
- a table with no label-based security policy.

You must specify the *label* name without the *policy* qualifier, rather than as *policy*.*label*, because the current security policy of the table is the only valid policy for any security label that protects data in the table.

The *column* cannot be of type IDSSECURITYLABEL.

When a user who holds appropriate label-based access privileges attempts to access a value in the protected column, the database server compares this label with the security credentials of the user, and allows or withholds access on the basis of this comparison.

You can use similar syntax in the ALTER TABLE MODIFY statement to associate an existing column of a protected table with an LBAC security label, or to drop the association between a protected column and its label. For more information, see Modify Column Security clause on page 144.

For the ALTER TABLE syntax to add an LBAC security policy to protect a table that has no security policy, or to drop the association of the current security policy with the table, see SECURITY POLICY Clause on page 154.

Example of adding a column protected by a security label

Suppose that protected table **SecuriTab** was created with the following schema:

```
CREATE TABLE SecuriTab
   (Col1 CHAR (18),
   Col2 INT,
   Col4 IDSSECURITYLABEL DEFAULT LabelRW)
   SECURITY POLICY company;
```

The table **SecuriTab** is protected with row-level granularity by the LBAC security policy **company**, **Col4** of type IDSSECURITYLABEL stores security label **LabelRW**. The security label **LabelRW** in the IDSSECURITYLABEL column

In the following statement, a user who holds the DBSECADM role adds a new CHAR(20) column **Col3** and provides row-level protection for its data by protecting that column with security label **Label23**:

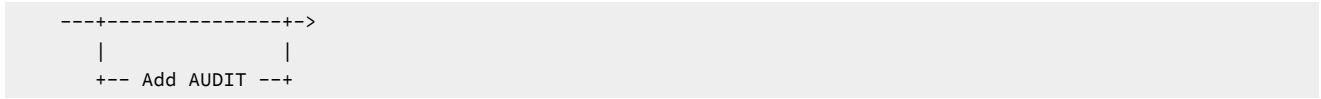
```
ALTER TABLE SecuriTab
   ADD (Col3 CHAR (20) BEFORE Col4 COLUMN SECURED WITH Label23);
```

Like the security label **Label23** in column **Col4**, the security label **Label23** must be a label of the current security policy, in this case the **company** policy.

ADD AUDIT Clause

Use the ADD AUDIT clause with the ALTER TABLE command to include a table in selective row-level auditing.

This syntax fragment is part of the ALTER TABLE statement on page 95.



When you alter a table with the ADD AUDIT clause, row-level audit events in that table are recorded when selective row-level auditing is turned on. Applying the ADD AUDIT attribute to a table by itself does not enable selective row-level auditing. This type of auditing is enabled when the ADTROWS parameter of the `adtcfg` file is set to 1 or 2 by using the onaudit -R command.

If selective row-level auditing is not enabled, the ADD AUDIT attribute on a table has no effect.

You must have RESOURCE or DBA privileges to run ALTER TABLE command with the ADD AUDIT clause.

ADD CONSTRAINT Clause

Use the ADD CONSTRAINT clause to specify a primary key, foreign key, referential, unique, or check constraint on a new or existing column or on a set of columns.

This syntax fragment is part of the ALTER TABLE statement on page 95.

(explicit id unique_139_Connect_42_addconc001) unique_139_Connect_42_addconc001(explicit id unique_139_Connect_42_addconc002)

unique_139_Connect_42_addconc002

ADD CONSTRAINT Clause

" ADD CONSTRAINT [] "

" (<Multiple-Column Constraint Format> []) "

For example, to add a unique constraint to the **fname** and **lname** columns of the **customer** table, use the following statement:

```
ALTER TABLE customer ADD CONSTRAINT UNIQUE (lname, fname);
```

To declare a name for the constraint, change the preceding statement by adding the CONSTRAINT keyword and an identifier for the constraint:

```
ALTER TABLE customer
   ADD CONSTRAINT UNIQUE (lname, fname) CONSTRAINT u_cust;
```

The name must be unique among the identifiers of constraints that are defined on the same table. If you define no name for the constraint, the database server assigns to the constraint a system-defined identifier, and stores this in the **sysconstraints.constrid** column of the system catalog.

The new constraint is enabled by default. To add a constraint that is not enabled, you can include the DISABLED keyword after the name of the constraint:

```
ALTER TABLE customer
   ADD CONSTRAINT UNIQUE (lname, fname) CONSTRAINT u_cust DISABLED;
```

Before you perform subsequent DML operations in which you want the constraint to be enforced. You can use the SET Database Object Mode statement to enable the disabled constraint.

When you do not specify a name for a new constraint, the database server provides one. You can find the name of the constraint in the **sysconstraints** system catalog table. For more information about the **sysconstraints** system catalog table, see the *HCL® Informix® Guide to SQL: Reference*.

Restrictions on constraints defined by ALTER TABLE

The following restrictions on the ADD CONSTRAINT clause (and on the MODIFY clause) affect constraints that the ALTER TABLE statement defines:

- When you add a constraint, the collating order must be the same as when the table was created.
- The ADD CONSTRAINT clause cannot define NULL or a NOT NULL constraints on columns of any data type. Only the MODIFY clause can define a NULL or a NOT NULL constraint on columns in existing tables.
- You cannot define primary key constraints, foreign key constraints, or unique constraints on RAW tables. You can, however, use the MODIFY clause of the ALTER TABLE statement to define a NOT NULL constraint or a NULL

constraint (but not both) on a column in a RAW table. For the syntax to add a NULL or NOT NULL constraint on a column in an existing table, see MODIFY Clause on page 139.

- You cannot place a unique constraint nor referential constraints on a BYTE or TEXT column.
- A check constraint on a BYTE or TEXT column can check only for IS NULL, IS NOT NULL, or LENGTH.
- By default, every IDSSECURITYLABEL column has an implicit NOT NULL constraint. You cannot, however, use the ADD CONSTRAINT clause to reference an IDSSECURITYLABEL column in the definition of a single-column constraint, nor as part of a multiple-column referential constraint or check constraint.

Multiple-Column Constraint Format

Use this option to assign one or more constraints to a column or to a set of columns in an existing table.

This closely resembles the syntax of the Multiple Column Constraint Format of the CREATE TABLE statement, but the optional INDEX DISABLED keywords are not valid (and return an error) in foreign key constraints that the CREATE TABLE statement defines. This syntax fragment is part of the ADD CONSTRAINT Clause on page 115.

(explicit id unique_174_Connect_42_multcoc001) unique_174_Connect_42_multcoc001(explicit id unique_174_Connect_42_multcoc002) unique_174_Connect_42_multcoc002(explicit id unique_174_Connect_42_multcoc003) unique_174_Connect_42_multcoc003(explicit id unique_174_Connect_42_multcoc004) unique_174_Connect_42_multcoc004(explicit id unique_174_Connect_42_multcoc005) unique_174_Connect_42_multcoc005(explicit id unique_174_Connect_42_multcoc006) unique_174_Connect_42_multcoc006(explicit id unique_174_Connect_42_multcoco007) unique_174_Connect_42_multcoco007

Multiple-Column Constraint Format

```
" { { { {  | NOT NULL  | NULL } | { UNIQUE |  DISTINCT  | PRIMARY KEY |  <REFERENCES Clause>  } } ( column ) |  <CHECK Clause>  <FOREIGN KEY Definition> } [  <Constraint>  ] |  <FOREIGN KEY Definition> [ CONSTRAINT constraint ]  INDEX DISABLED  } "
```

FOREIGN KEY Definition

```
" FOREIGN KEY ( column )  <REFERENCES Clause>  "
```

Element	Description	Restrictions	Syntax
column	A column on which the constraint is placed	No more than 16 columns	Identifier on page 1231
constraint	The name of a disabled foreign-key constraint	Must be unique among the names of indexes and constraints in the database	Identifier on page 1231

As in the CREATE TABLE statement, the Multiple-Column Constraint format for ALTER TABLE differs from the Single-Column Constraint format by requiring the FOREIGN KEY keywords before the REFERENCES clause when you specify a foreign key constraint. In addition, as its name implies, the Multiple-Column format can specify a list of columns as the scope of the new constraint, but this syntax is also valid with a single column.

For information about the INDEX DISABLED keyword option, see Using the INDEX DISABLED keywords in a foreign key definition on page 120.

A multiple-column constraint has these cardinality and size restrictions:

- It can specify no more than 16 column names.
- The maximum total length of the list of columns depends on the page size, according to this formula:

```
MAXLength = (((PageSize - 93)/5) -1)
```

  - For a page size of 2K, the total length cannot exceed 390 bytes.
  - For a page size of 16K, the total length cannot exceed 3257 bytes.

Here the slash (/) symbol represents integer division.

The statement fails with an error if you specify both a NOT NULL constraint and a NULL constraint on the same column, or if you define a NOT NULL constraint on a column whose default value is NULL.

You cannot define a NULL constraint on a column whose data type is LIST, MULTISET, SET, or IDSSECURITYLABEL.

If the constraint is on a set of columns that includes a column that stores encrypted data, Informix® cannot enforce the constraint. You can declare a name for the constraint and set its mode with Constraint Definition on page 108.

If the ALTER TABLE ADD CONSTRAINT statement defines more than one referential constraints on the same table, each constraint requires its own REFERENCES clause, so that options like ON DELETE CASCADE can be specified (or omitted) for each individual constraint, rather than applied to all of the constraints.

If the database server implicitly creates an index on the same non-opaque column or set of columns as the referential constraint, distribution statistics are automatically calculated on the specified column, or on the lead column of a multiple-column constraint.

These distribution statistics are equivalent to distributions created by the UPDATE STATISTICS statement in HIGH mode, and are available to the query optimizer when it designs query plans for the table on which the new constraint was created. See also the section Automatic Calculation of Distribution Statistics on page 261 in the description of the CREATE INDEX statement for additional information about statistical distributions that are calculated when an index or constraint is created on an existing table.

Example of adding a multiple-column referential constraint

In the **stores_demo** database, the **stock** table contains the **stock_num** column as its primary key. An enabled referential constraint **aa** on the **catalog** table has columns **catalog.stock_num** and **catalog.manu_code** as its multiple-column foreign-key, referencing the **stock** table. The following example drops that existing foreign-key constraint, and add a new constraint called **ai**, in DISABLED object mode, that disables the associated system-generated index on those columns:

```
ALTER TABLE catalog DROP CONSTRAINT aa;

ALTER TABLE catalog ADD CONSTRAINT
   (FOREIGN KEY (stock_num, manu_code) REFERENCES stock
   CONSTRAINT ai DISABLED INDEX DISABLED);
```

The parentheses that delimit the definition of the new constraint are optional, the following ALTER TABLE statement is equivalent:

```
ALTER TABLE catalog ADD CONSTRAINT
   FOREIGN KEY (stock_num, manu_code) REFERENCES stock
   CONSTRAINT ai DISABLED INDEX DISABLED;
```

The first DISABLED sets the object more of the constraint. The last DISABLED sets the object mode of the index.

Creating foreign-key constraints when an index exists on the referenced table

By default, the database server automatically validates enabled referential constraints when the ADD CONSTRAINT or MODIFY option to the ALTER TABLE statement includes the REFERENCES keyword to define a foreign-key constraint. You might be able to save time during validation of the new foreign-key constraint, if the referenced table already has a unique index or a primary-key constraint on the column (or on the set of columns) corresponding to the key of the referential constraint.

The database server makes a cost-based decision on how to validate the foreign-key constraint. The index-key algorithm might be faster in many contexts, because it validates the new constraint by scanning only the index values, rather than by scanning all the rows in the table.

The database server can consider using the index-key algorithm to validate the foreign-key constraint that it creates, but only if all of the following conditions are satisfied:

- The ALTER TABLE statement is creating only one foreign-key constraint.

  If this is the case, the database server needs to check individual values for only the column on which the foreign-key constraint is being created. Validating two foreign-key constraints at the same time would require two indices to be used on the same scan, which is not supported.
- The statement is not also creating or enabling a CHECK constraint.

  If the ALTER TABLE statement is creating more than one constraint, validating CHECK constraints requires that every row be checked, rather than individual values. In that case, the index-key algorithm cannot be used for validating the foreign-key constraint.
- The statement that creates the foreign-key constraint does not also change the data type of any existing column in the same table.

  If the ALTER TABLE statement that creates the foreign-key constraint includes a MODIFY clause that changes the data type of any column, the database server does not consider an index-scan execution path for validating the constraint.
- The foreign-key columns do not include user-defined data types (UDTs) or built-in opaque data types.

To make the fast index-key algorithm as efficient as possible, it eliminates all the inefficiencies of executing routines associated with user-defined or opaque data types, such as the BOOLEAN and LVARCHAR built-in opaque types.

- The mode of the new foreign-key constraint is not DISABLED.

  If it is disabled, then no constraint-checking algorithm is needed, because no checking for referential integrity violations occurs.

- The table is not associated with an active violation table.

  Violations tables require that at the time of checking, every row that does not satisfy the new constraint must be inserted into the violation table. Scanning every row for violations prevents the database server from using the faster index-key algorithm that skips duplicate rows.

Except in the case of one or more violating rows, the ALTER TABLE ADD CONSTRAINT or ALTER TABLE MODIFY statement can create and validate a foreign-key constraint when some of these requirements are not satisfied, but the database server will not consider using the index-key algorithm to validate the foreign-key constraint. The additional validation costs for scanning the entire table are generally proportional to the size of the table. These costs can be substantial for very large tables.

When you create a self-referencing foreign-key constraint, whose REFERENCING clause specifies the same table on which the constraint is defined, the database server can consider an index-key algorithm for validating referential-integrity, if all of the conditions listed above are satisfied.

Using the INDEX DISABLED keywords in a foreign key definition

By including the optional INDEX DISABLED keywords when you define a foreign key constraint, you prevent DML operations on the table from using the index that the database server associates with the foreign key. If you include the INDEX DISABLED keywords, they must be the last specification in the ALTER TABLE statement.

Defining foreign key constraints in the ALTER TABLE statement

To add a foreign key constraint, you must have the References privilege on either the referenced columns or the child table. If you own the parent table or have the Alter privilege on the parent table, you can create a foreign key constraint on that table and specify yourself as the owner of the constraint. When you hold the DBA privilege, you can create foreign key constraints for other users.

When the ALTER TABLE ADD CONSTRAINT statement places a foreign key constraint on a column or on a set of columns that reference a child table, and no referential constraint or user-defined index already exists on that column or on that set of columns, the database server creates an internal B-tree index on the specified column or set of columns. If a user-created index already exists on that column or set of columns, the constraint shares the existing index.

If the ALTER TABLE ADD CONSTRAINT statement defines more than one foreign key constraint on the same table, each constraint requires its own REFERENCES clause, and the INDEX DISABLED keywords can be specified (or omitted) for each constraint.

This INDEX DISABLED option is valid in ALTER TABLE ADD CONSTRAINT statements issued from updatable secondary servers in cluster environments.

Circumstances where a foreign key index can reduce performance

Although referential constraints protect data integrity, in some contexts the user-defined or system-generated B-tree index that the database server associates with a foreign key constraint can reduce the efficiency of data manipulation operations on tables that are very large. If there are no deletes from the parent table, the index is not used to look up rows in the child table for cascading deletes. If no queries use this index on the child table. In this scenario, the index is simply not needed, but it imposes unnecessary overhead in operations that update, delete, or insert rows into the child table. A data warehousing application on a child table with millions of rows could require fewer resources if the index that corresponds to the foreign key constraint were disabled.

In these cases, the INDEX DISABLED keyword option to the ALTER TABLE ADD CONSTRAINT statement offers a mechanism for defining a foreign key constraint but avoiding the overhead of the large associated b-tree index.

When you include the INDEX DISABLED keywords at the end of the constraint definition, the database server disables the system-generated index, if no appropriate user-defined index already exists. If a user-defined index on the foreign key column (or set of columns) of the child table already exists, the database server disables that index. Subsequent DML operations on the child table are accomplished without an index, and minimal system resources are needed for maintenance and storage of the disabled index.

Effects of the INDEX DISABLED keywords

When you include the INDEX DISABLED keywords at the end of the constraint definition, the database server disables the system-generated index, if no appropriate user-defined index already exists. If a user-defined index on the foreign key column (or set of columns) of the child table already exists, the database server disables that index. Subsequent foreign key enforcement during DML operations on the child or parent table are accomplished without this disabled index, and minimal system resources are needed for maintenance and storage of the disabled index.

These are the actions of the database server when you successfully add a foreign key constraint with the INDEX DISABLED option:

- The index associated with the foreign key constraint is identified.
- That index is disabled, and marked as disabled in the **sysobjstate** table of the system catalog.
- The physical index is dropped from the database.
- The **sysfragments** system catalog table is updated to show no storage allocation for that index.

The INDEX DISABLED keywords have no effect on foreign key constraint that you define. The database server enforces that constraint, and issues an error if any subsequent operation on the child table or on the parent table violates the specified foreign key constraint.

The following restrictions apply to the INDEX DISABLED keywords in constraint definitions:

- The INDEX DISABLED option is valid only in foreign key definitions.
- Only the ALTER TABLE ADD CONSTRAINT statement supports this syntax. The CREATE TABLE or ALTER TABLE MODIFY COLUMN statements return an exception if a foreign key constraint definition includes the INDEX DISABLED keywords.
- If the index used by the foreign key is in use by another constraint, the database server returns an error.
- If you include the DISABLED keyword in the constraint definition to disable the foreign key constraint, the database server returns an error if you also specify the INDEX DISABLED keywords, as in the following example.

```
ALTER TABLE child ADD
  CONSTRAINT(FOREIGN KEY(x1) REFERENCES parent(c1)
     CONSTRAINT cons_child_x1 DISABLED INDEX DISABLED);
```

To correct the error in ALTER TABLE ADD CONSTRAINT example above, you must either drop the first `DISABLED` keyword, or else drop the `INDEX DISABLED` keywords.

Example of creating a foreign key constraint with INDEX DISABLED

Suppose that the **parent** table and the **child** table in the following example have a primary key and foreign key dependency, and that the data stored in these tables satisfies the following conditions:

- The **parent** table has only a few rows.
- The **child** table has millions of rows.
- The foreign key columns in the **child** table have only few distinct possible values, based on the primary key of the **parent** table.

This example shows how to use the INDEX DISABLED keyword option of the ALTER TABLE ADD CONSTRAINT statement.

```
CREATE TABLE parent(c1 INT, c2 INT, c3 INT);
CREATE UNIQUE INDEX idx_parent_c1 ON parent(c1);
ALTER TABLE parent ADD
   CONSTRAINT PRIMARY KEY(c1)
      CONSTRAINT cons_parent_c1;
CREATE TABLE child(x1 INT, x2 INT, x3 VARCHAR(32));
CREATE INDEX idx_child_x1 ON child(x1);

ALTER TABLE child ADD
   CONSTRAINT(FOREIGN KEY(x1) REFERENCES parent(c1)
      CONSTRAINT cons_child_x1 INDEX DISABLED);
```

In the example above,

- **cons_parent_c1** is a primary key constraint on the **parent** table,
- **cons_child_x1** is a foreign key constraint on the **child** table,
- **idx_parent_c1** is a unique index shared by the **cons_parent_c1** constraint,
- and **idx_child_x1** is an index shared by the **cons_child_x1** constraint.

Data manipulation language operations like UPDATE, DELETE, INSERT, and MERGE on the child table cannot use the **idx_child_x1** index that is shared by with the foreign key constraint, because that index is now disabled.

For some tables that have a primary key and foreign key dependencies, however, the query optimizer might choose other indexes on the child table, based on WHERE clause predicates, in execution plans.

As indicated above, use of the `INDEX DISABLED` option in the foreign key definition can improve performance only when the child table is very large, typically in the context of data warehouse applications. This syntax option is not recommended for DML operations on small tables.

Creating foreign-key constraints in NOVALIDATE modes

The ALTER TABLE ADD CONSTRAINT statement can create an enabled or filtering foreign-key constraint in a NOVALIDATE mode. The NOVALIDATE constraint modes prevent the database server from verifying that the foreign-key value in every row matches a primary-key value in the referenced table while the referential constraint is being created.

Use this syntax to create an enabled or filtering foreign-key constraint in NOVALIDATE mode:

ADD FOREIGN KEY CONSTRAINT in NOVALIDATE mode

```
ALTER TABLE { table | synonym } ADD CONSTRAINT { <Foreign key constraint in NOVALIDATE mode> | ( <Foreign key constraint in NOVALIDATE mode> ) }
```

Foreign key constraint in NOVALIDATE mode

> " FOREIGN KEY(*fk_col*)REFERENCES *pk_tab* [(*pk_col*)] "
>
> " { CONSTRAINT*constraint* | ON DELETE CASCADE } "
>
> " [{ ENABLED | FILTERING { WITHOUT ERROR | WITH ERROR } (explicit id) }] (explicit id) NOVALIDATE "

Element	Description	Restrictions	Syntax
constraint	Name declared here for the constraint	Must be unique among constraint and index names in the database	Identifier on page 1231
fk_col	Foreign-key column for *constraint*	Must exist in the child table	Identifier on page 1231
pk_col	Primary-key column in the referenced table	Must exist in the referenced table	Identifier on page 1231
pk_tab	Name of the referenced table	Must exist in the current database	Identifier on page 1231
table, *synonym*	Table on which *constraint* is placed	Must exist in the current database	Identifier on page 1231

Usage

This diagram omits the `DISABLED` keyword. Because disabled constraints are not checked for violations, the `NOVALIDATE` keyword is unnecessary in that case.

If no column or list of columns immediately follows the `REFERENCES` keyword, the default column (or columns) is the primary key of the *pk_tab* table. If *pk_tab* and the *table* or *synonym* specify the same table, then the constraint is self-referencing, and there is no default primary-key column.

If you declare no constraint name, the database server generates an identifier for the new constraint that it registers in the **sysconstraints** and **sysobjstate** system catalog tables.

The ALTER TABLE ADD CONSTRAINT statement supports the NOVALIDATE mode for referential constraints as a mechanism for bypassing the data-integrity check when creating an enabled or filtering referential constraint.

Circumstances where NOVALIDATE modes can improve performance

Although referential constraints protect data integrity, in some contexts a database table that you are moving to a new database server instance is known to be free of referential integrity violations. For foreign-key constraints on large tables, the time required to validate the constraint can be substantial. If a table with a million rows is moving from an OLTP environment to a data warehousing environment, validating the foreign key in the target environment might increase the time required for migration by orders of magnitude.

For example, you can drop the foreign-key constraints on the large table, and then re-create those constraints in an ENABLED NOVALIDATE mode or in a FILTERING NOVALIDATE mode immediately before you migrate the large table to the target database environment. The cost of the ALTER TABLE ADD CONSTRAINT operation that re-creates the foreign-key constraint will be relatively small, because it bypasses validation of every row for each referential constraint. Because the NOVALIDATE mode does not persist beyond the ALTER TABLE operation that created the constraints, the table arrives in the warehousing environment with those constraints in an ENABLED or FILTERING mode, protecting the referential integrity of the data in subsequent DML operations.

Restrictions on using the NOVALIDATE keyword

The ALTER TABLE ADD CONSTRAINT statement is the only DDL context where the `NOVALIDATE` keyword is valid when a foreign-key constraint is being created. You cannot, for example, create a foreign-key constraint in a NOVALIDATE mode in any of the following SQL statements:

- CREATE TABLE statements
- CREATE TEMP TABLE statements
- SELECT INTO TABLE statements.

You can use the ALTER TABLE ADD CONSTRAINT statement to create an enabled constraint on an existing table in NOVALIDATE mode only if all of the following conditions are satisfied:

- The constraint that you are adding is a foreign-key constraint. If you create multiple constraints in an ALTER TABLE statement that includes the `NOVALIDATE` keyword, all must be foreign-key constraints.
- In ALTER TABLE statements, the `NOVALIDATE` keyword is valid only in the ADD CONSTRAINT FOREIGN KEY option.
- The `NOVALIDATE` keyword is not valid for constraints that ALTER TABLE creates in DISABLED mode.

The ALTER TABLE statement fails with an error if the constraint definition includes the `NOVALIDATE` keyword in any of the following syntax contexts:

- ALTER TABLE ADD *column* statements.
- ALTER TABLE INIT statements
- ALTER TABLE MODIFY statements.

The only other DDL statement where the `NOVALIDATE` keyword is valid is the SET CONSTRAINTS option to the SET Database Object Mode statement. While the SET CONSTRAINTS statement is running, it can change the mode of an existing foreign-key constraint to any of these NOVALIDATE constraint modes:

- ENABLED NOVALIDATE mode
- FILTERING WITH ERROR NOVALIDATE mode
- FILTERING WITHOUT ERROR NOVALIDATE mode.

For more information, see the SET CONSTRAINTS statement on page 782.

Establishing NOVALIDATE modes as the default

Both the SET ENVIRONMENT NOVALIDATE ON statement of SQL and the **`dbimport -nv`** command for loading databases can override any foreign-key constraint mode (except `DISABLED`) that ALTER TABLE ADD CONSTRAINT or SET CONSTRAINTS statement specifies, if that constraint mode specification omits the `NOVALIDATE` keyword.

- The scope of the SET ENVIRONMENT NOVALIDATE ON statement is subsequent ALTER TABLE ADD CONSTRAINT and SET CONSTRAINTS statements in the same user session.
- The scope of the **`dbimport -nv`** command is the ALTER TABLE ADD CONSTRAINT and SET CONSTRAINTS statements in the **`.sql`** file of the exported database, whose path name is specified in the same **`dbimport`** command.

Just as when `NOVALIDATE` is specified explicitly in ALTER TABLE ADD CONSTRAINT and SET CONSTRAINTS statements, during the session that issued the SET ENVIRONMENT NOVALIDATE ON statement the default NOVALIDATE mode for foreign-key constraints persists

- only while the foreign-key constraint is being created,
- or only while an existing constraint is being reset to ENABLED or to FILTERING mode.

When the ADD CONSTRAINT or SET CONSTRAINTS statement completes execution without validating the foreign-key constraint, no record of the transient NOVALIDATE mode is created in the **sysobjstate** system catalog table. Instead, the database server registers the mode of the new or updated foreign-key constraint as ENABLED or as FILTERING in the **sysobjstate.state** column of its row in the system catalog.

Example of a constraint created in NOVALIDATE mode

The following DDL statements create a table called **parent** and define a unique index and a primary-key constraint on column **c1** of that table:

```
CREATE TABLE parent(c1 INT, c2 INT, c3 INT);
CREATE UNIQUE INDEX idx_parent_c1 ON parent(c1);
ALTER TABLE parent ADD CONSTRAINT
   PRIMARY KEY(c1) CONSTRAINT cons_parent_c1;
```

The next statements create another table, called **child**, whose first column is of the same data type as the primary-key column of the **parent** table, and define an enabled foreign-key constraint, called **cons_child_x**, on the **child** table:

```
CREATE TABLE child(x1 INT, x2 INT, x3 VARCHAR(32));
ALTER TABLE child
   ADD CONSTRAINT (FOREIGN KEY(x1)
       REFERENCES parent(c1) CONSTRAINT cons_child_x1);
```

Suppose that subsequent DML operations (not shown) populate the **parent** table and the **child** table with rows of data. At some point, the workflow requires the data to be moved from its OLTP production environment to another database for processing by business-analytics applications.

If at this point the data set in the **child** table contains a large number of rows, a significant cost of importing the **child** table to its new database will be validating the **cons_child_x1** referential constraint. To avoid this cost. the following statement drops that constraint:

```
ALTER TABLE child DROP CONSTRAINT cons_child_x1;
```

After the **child** table has been exported to its new environment, the following statement can re-create a new constraint of the same name on the **child** table without checking every row for referential-integrity violations:

```
ALTER TABLE child
   ADD CONSTRAINT (FOREIGN KEY(x1)
      REFERENCES parent(c1)
         CONSTRAINT cons_child_x1 NOVALIDATE);
```

By default, the new **cons_child_x1** referential constraint is in ENABLED mode after the ALTER TABLE statement completes execution.
Like the optional parentheses that can enclose various ALTER TABLE ADD CONSTRAINT syntax options, the parentheses enclosing the "Foreign key constraint in NOVALIDATE mode" syntax segment are optional. The next statement has the same effect as the previous `NOVALIDATE` example:

```
ALTER TABLE child
   ADD CONSTRAINT FOREIGN KEY(x1)
      REFERENCES parent(c1)
         CONSTRAINT cons_child_x1 NOVALIDATE;
```

Including the parentheses might make the logical structure clearer to human readers, but those delimiters have no effect on how the database server processes this ALTER TABLE ADD CONSTRAINT NOVALIDATE statement without validating each record.

Adding a Primary-Key or Unique Constraint

When you place a primary-key or unique constraint on a column or set of columns, those columns must contain unique values. The database server checks for existing constraints and indexes:

- If a user-created unique index already exists on that column or set of columns, the constraint shares the index.
- If a user-created index that allows duplicates already exists on that column or set of columns, the database server returns an error.

  In this case, you must drop the existing index before adding the primary-key or unique constraint.
- If a referential constraint already exists on that column or set of columns, the duplicate index is upgraded to unique (if possible) and the index is shared.
- If no referential constraint or user-created index exists on that column or set of columns, the database server creates an internal B-tree index on the specified columns.

When you place a referential constraint on a column or set of columns, and an index already exists on that column or set of columns, the index is shared.

If you own the table or have the Alter privilege on the table, you can create a check, primary-key, or unique constraint on the table and specify yourself as the owner of the constraint. To add a referential constraint, you must have the References privilege on either the referenced columns or the referenced table. When you have the DBA privilege, you can create constraints for other users.

Recovery from Constraint Violations

If you use the ADD CONSTRAINT clause to add a constraint in the enabled mode, you receive an error message because existing rows would violate the constraint. For a procedure to add the constraint successfully, see Adding a Constraint That Existing Rows Violate on page 148.

ADD TYPE Clause

Use the ADD TYPE clause to convert a table that is not based on a named ROW data type into a typed table.

This clause is an extension to the ANSI/ISO standard for SQL.

ALTER TABLE { *table* | *synonym* } <ADD TYPE Clause>

ADD TYPE Clause

" **ADD TYPE** *row_type* "

Element	Description	Restrictions	Syntax
row_type	Identifier of an existing named ROW data type for this table	The *row_type* fields must exactly match the existing columns in their data types, order, and number	Identifier on page 1231
synonym	Synonym for the table that becomes a typed table	Synonym and its table must exist, and **USETABLENAME** must not be set	Identifier on page 1231
table	Identifier of the table that becomes a typed table	Must exist in the current database	Identifier on page 1231

Usage

If the new ROW type is a subtype of an existing named ROW type, you must hold the UNDER privilege for that named ROW type.

When you use the ADD TYPE clause, you assign the specified named ROW data type to a table whose columns must exactly match the fields of that ROW data type

- in their names,
- in their individual data types,
- and in their order.

In addition to the requirements common to all ALTER TABLE operations (namely DBA privilege on the database, Alter privilege on the table, and ownership of the table), all of the following must be also true when you use the ADD TYPE clause to convert an untyped table to the specified named ROW data type:

- The named ROW data type is already registered in the database.
- You hold the Usage privilege on the named ROW data type.
- There must be a 1-to-1 correspondence between the ordered set of column data types of the untyped table and the ordered set of field data types of the named ROW data type.
- The table cannot be a fragmented table that has **rowid** values.

You cannot combine the ADD TYPE clause with any clause that changes the schema of the table. No other ADD, DROP, or MODIFY clause is valid in the same ALTER TABLE statement that has the ADD TYPE clause. The ADD TYPE clause does not allow you to change column data types. (To change the data type of a column, use the MODIFY clause in a separate ALTER TABLE statement from the ADD TYPE operation.)

ALTER TABLE operations on typed tables

Most options of the ALTER TABLE statement are restricted to untyped tables.

The database server supports only the following storage, locking, and constraint syntax options when *table_object* is the name or synonym of a typed table. As noted below, however, for some CONSTRAINT options in a typed-table inheritance hierarchy

- the scope of the operation includes all subtables of *table_object*,
- or the operation is not supported if *table_object* is a subtable.

ALTER TABLE options for typed tables

(explicit id unique_147_Connect_42_typt002) unique_147_Connect_42_typt002(explicit id unique_147_Connect_42_typt003) unique_147_Connect_42_typt003(explicit id unique_147_Connect_42_typt004) unique_147_Connect_42_typt004(explicit id unique_147_Connect_42_typt005) unique_147_Connect_42_typt005(explicit id unique_147_Connect_42_typt006) unique_147_Connect_42_typt006 **ALTER TABLE***table_object* <Typed-table options>

Typed-Table options

" { | <ADD CONSTRAINT Clause> [] | <DROP CONSTRAINT Clause> [] | [] <MODIFY NEXT SIZE Clause> [] | [] <LOCK MODE Clause> [] } "

Two additional considerations apply to typed tables of named ROW types that are part of inheritance hierarchies:

- For subtables, ADD CONSTRAINT and DROP CONSTRAINT are not valid on inherited constraints.
- For supertables, ADD CONSTRAINT and DROP CONSTRAINT propagate to all subtables.

Example of the ALTER TABLE . . . ADD TYPE statement

The following example changes an untyped table into a typed table by the following steps:

- Use the CREATE TABLE statement to create an table that will become the typed table.
- Use the CREATE ROW TYPE statement to create a named ROW type with fields exactly matching the untyped table schema.
- Use the ALTER TABLE . . . ADD TYPE statement to change the untyped table to a typed table.

```
CREATE TABLE postal(
   name     VARCHAR(30),
   address  VARCHAR(20),
   city     VARCHAR(20),
   state    CHAR(2),
   zip      INTEGER,
);

. . .

CREATE ROW TYPE postal_t
(
   name     VARCHAR(30),
   address  VARCHAR(20),
   city     VARCHAR(20),
   state    CHAR(2),
   zip      INTEGER,
);
```

```
. . .

ALTER TABLE postal ADD TYPE postal_t;
```

You could achieve the same result if the order of the first two DDL statements were reversed.

In that case, however, the ALTER TABLE statement could be avoided, if instead of defining columns individually, the CREATE TABLE instead included the OF TYPE clause to use the **postal_t** ROW type as a template for the six fields of a single named ROW type column:

```
CREATE TABLE postal OF TYPE postal_t;
```

By either path, table **postal** becomes a typed table of ROW type **postal_t**.

In the example above, as elsewhere in the documentation of typed tables, the database server does not require any similarity between the SQL identifier of the table and the SQL identifier of the ROW type. Table objects and named ROW data types have separate and uncorrelated name spaces, but mnenomic names can make SQL code examples easier for some human readers to understand.

DROP AUDIT Clause

Use the DROP AUDIT clause with the ALTER TABLE command to remove it from the set of tables that is audited when selective row-level auditing is enabled.

This syntax fragment is part of the ALTER TABLE statement on page 95.

```
   ---+---------------+->
      |               |
      +-- Drop AUDIT -+
```

The DROP AUDIT clause affects only tables that have been flagged for inclusion in selective row-level auditing. If you have not created or altered the table with the WITH AUDIT clause or ADD AUDIT clause, it is not necessary to use DROP AUDIT to exclude it from the set of tables that are audited at the row level.

Removing the AUDIT attribute from a table does not disable or change selective row-level auditing of other tables in the database.

You must be a DBSSO to run the ALTER TABLE command with the DROP AUDIT clause.

DROP CONSTRAINT Clause

Use the DROP CONSTRAINT clause to destroy an existing constraint whose name you specify.

This syntax fragment is part of the ALTER TABLE statement on page 95.

DROP CONSTRAINT Clause

" DROP CONSTRAINT "

" (constraint) "

Element	Description	Restrictions	Syntax
constraint	Constraint to be dropped	Must exist in the database	Identifier on page 1231

Usage

To drop an existing constraint, specify the `DROP CONSTRAINT` keywords and the identifier of the constraint. To drop multiple constraints on the same table, the constraint names must be in comma-separated list that is delimited by parentheses.

The constraint that you drop can have an ENABLED, DISABLED, or FILTERING mode.

Here is an example of dropping a constraint:

```
ALTER TABLE manufact DROP CONSTRAINT con_name;
```

The following example destroys both a referential constraint and a check constraint that had been defined on the **orders** table:

```
ALTER TABLE orders DROP CONSTRAINT (con_ref, con_check);
```

The Informix® implementation of SQL includes no DROP CONSTRAINT statement. This clause of the ALTER TABLE statement, however, provides functionality that one might expect of the DROP CONSTRAINT statement, if that statement existed.

The DROP TABLE statement implicitly drops all constraints on the specified table when it destroys that table.

Retrieving constraint names

The DROP CONSTRAINT clause requires the identifier of the constraint. If no name was declared when the constraint was created, the database server generated the identifier of the new constraint. You can query the **sysconstraints** system catalog table for the name and the owner of a constraint. For example, to find the name of the constraint placed on the **items** table, you can issue the following statement:

```
SELECT constrname FROM  sysconstraints
   WHERE tabid = (SELECT tabid FROM systables
      WHERE tabname = 'items');
```

Dependencies between constraints

When you drop a primary-key constraint or a unique constraint that has a corresponding foreign key, any associated referential constraints are also dropped.

For example, in the **stores_demo** database, there is a primary-key constraint on the **order_num** column in the **orders** table. A corresponding foreign-key constraint is also defined on the **order_num** column in the **items** table. These constraints define a referential relationship between the **order_num** columns in both tables.

Suppose that you run the `ALTER TABLE orders DROP CONSTRAINT` statement to drop the primary-key constraint on the **order_num** column in the **orders** table. Because this referential-integrity relationship between the two tables can no longer be enforced without the primary-ley constraint, the database server takes these actions if the ALTER TABLE statement in this example succeeds:

- It destroys the specified primary-key constraint on the **order_num** column in the **orders** table.
- It also destroys the corresponding referential constraint on the **order_num** column in the **items** table.
- It deletes from the system catalog all references to the primary key constraint on the **orders** table, or to the referential constraint on the **items** table.

System catalog effects of dropping constraints

The database maintains information about existing constraints in these system catalog tables:

- **sysconstraints** (all constraints)
- **sysobjstate** (all constraints)
- **syschecks** (check constraints)
- **syscoldepend** (check constraints and NOT NULL constraints)
- **syscheckudrdep** (check constraints that UDRs reference)
- **sysreferences** (referential constraints)
- **sysindices** (referential, primary-key, or unique constraints that have no corresponding Index entry in **sysindices**)

After the DROP CONSTRAINT clause successfully destroys a constraint, the database server deletes or updates at least one row in one or more of the above tables.

Data type considerations

By default, every IDSSECURITYLABEL column has an implicit NOT NULL constraint, but the DROP CONSTRAINT clause cannot reference columns of type IDSSECURITYLABEL.

You can include the DROP CONSTRAINT clause in ALTER TABLE operations on typed tables of ROW data types, but with these restrictions for table hierarchies.

- For a typed table with no supertable, DROP CONSTRAINT propagates to its child tables.
- For the child table of a supertable, DROP CONSTRAINT fails for inherited constraints.

For example, suppose ROW type **people_t** exists in the database. The following DDL statements respectively

- create a typed table called **MyPeople**,
- add a UNIQUE constraint on **MyPeople** called **very_unique** on all the fields of the ROW type column,
- and create a child table called **LittlePeople**:

```
CREATE TABLE IF NOT EXISTS MyPeople OF TYPE (people_t);
ALTER TABLE MyPeople ADD CONSTRAINT UNIQUE (people_t.*)
   CONSTRAINT very_unique;
CREATE TABLE IF NOT EXISTS LittlePeople OF TYPE (people_t)
   UNDER MyPeople;
```

Because **LittlePeople** inherited the **very_unique** constraint from its parent supartable **MyPeople**, the following ALTER TABLE DROP CONSTRAINT statement fails:

```
ALTER TABLE LittlePeople
   DROP CONSTRAINT very_unique; --cannot drop an inherited constraint
```

But because **MyPeople** is a typed table with no parent, the following statement destroys the **very_unique** constraint instances on both the **MyPeople** and the **LittlePeople** tables within their table hierarchy:

```
ALTER TABLE MyPeople DROP CONSTRAINT very_unique;
```

Restoring a referential constraint

For some operations, such as relocating a table to another database, you might require that a referential constraint temporarily have no effect on its table. After you complete that work without the constraint, however, the referential integrity of the database typically requires that the functionality of the constraint be restored. One possible option is this:

- Use the DROP CONSTRAINT clause to destroy the constraint.
- Complete the tasks that need to avoid the effects of the constraint.
- Use ALTER TABLE ADD CONSTRAINT to re-create the constraint.

For a very large table that already conforms to the dropped foreign-key or check constraint, using the NOVALIDATE option to the ALTER TABLE ADD CONSTRAINT statement can avoid the substantial cost of using a full-table scan to validate the constraint while it is being re-created. This NOVALIDATE option requires using the Multi-Column Constraint Format syntax to define the constraint.

Similarly, rather than destroying a referential constraint on a large table, you can temporarily disable it, and then complete tasks that require fewer resources with the constraint disabled. To resume enforcement of the constraint, you can use the SET CONSTRAINTS option of the SET Database Object Mode statement to reset the object mode to ENABLED NOVALIDATE, or to FILTERING WITH ERROR NOVALIDATE or to FILTERING WITHOUT ERROR NOVALIDATE. In each of these constraint modes, the `NOVALIDATE` keyword avoids the overhead of validating the constraint while its mode is being reset.

DROP Column Clause

Use the DROP Column clause to remove one or more columns from the schema of a table.

This syntax fragment is part of the ALTER TABLE statement on page 95.

DROP Column Clause

" DROP "

" { (column) | column } "

Element	Description	Restrictions	Syntax
column	Name of a column to be dropped	Must exist in the table. No fragment expression can reference the column, and it cannot be the last column in the table.	Identifier on page 1231

You cannot issue an ALTER TABLE DROP statement that would drop every column from the table. At least one column must remain in the table.

You cannot drop a column that is part of the fragmentation key of a fragmentation strategy.

A column that is protected by a security label can be dropped by the ALTER TABLE DROP statement, but the user must be DBSECADM and must also hold the usual CONNECT, RESOURCE, and ALTER access privileges for modifying the schema of the table.

How Dropping a Column Affects Constraints

When you drop a column, all constraints on that column are also dropped, unless the same ALTER TABLE statement also adds the same column in the same order within the table schema.

Dropping a column has these effects on its constraints:

- All single-column constraints are dropped.
- All referential constraints that reference the column are dropped.
- All check constraints that reference the column are dropped.
- If the column is part of a multiple-column primary-key or unique constraint, the constraints that are placed on the multiple columns are also dropped. This action, in turn, triggers the dropping of all referential constraints that reference the multiple columns.
- The **syscolumns** and **sysconstraints** system catalog tables are updated to remove all the rows corresponding to the dropped columns and the dropped constraints.

Because any constraints that are associated with a column are dropped when the column is dropped, the structure of other tables might also be altered when you use this clause. For example, if the dropped column is a unique or primary key that is referenced in other tables, those referential constraints also are dropped. Therefore, the structure of those other tables is also altered.

Dropping columns without changing the table schema

The same ALTER TABLE statement can include both a DROP Column clause that drops a column (or drops several columns), followed by an ADD Column clause that adds columns of the same name, data type, and ordinal position among the columns of the table. In this case, the database server takes no action to reorganize the schema of the table, or to drop constraints that are defined on the columns that were dropped and restored in the same ALTER TABLE statement. That is, nothing happens to the table or to its data.

For example, suppose that table **UnsTable** is created with five columns:

```
CREATE TABLE IF NOT EXISTS UnsTable (
   col1 CHAR(18),
   col2 INT NOT NULL,
   col3 CHAR(32) UNIQUE,
   col4 INT NOT NULL,
   col5 DATETIME YEAR TO MONTH);
```

The following ALTER TABLE statement instructs the database server to drop both INTEGER columns **col2** and **col4**, and also to add two columns with the same names and data type:

```
ALTER TABLE UnsTable (
   DROP (col2, col4)
   ADD (col2 INT NOT NULL BEFORE col3,
        col4 INT NOT NULL BEFORE col5);
```

This ALTER TABLE statement has no effect on the schema or the data in the **UnsTable** table, nor on any existing constraints that reference those columns, because the database server takes no action. Therefore, DDL statements that resemble the example that are generated by analytic tools do not affect the database server or the data.

When you need to reorganize a table and drop constraints by replacing one or more columns with new columns that have the same names, the same data types, and the same ordinal positions within the table schema, run two separate ALTER TABLE statements, as in the following example:

```
ALTER TABLE UnsTable (
   DROP (col2, col4);

ALTER TABLE UnsTable (
   ADD (col2 INT NOT NULL BEFORE col3,
        col4 INT NOT NULL BEFORE col5);
```

The ALTER TABLE DROP statement drops any existing constraints that reference columns **col2** or **col4** of the **UnsTable** table.

How Dropping a Column Affects Triggers

In general, when you drop a column from a table, the triggers based on that table remain unchanged. If the column that you drop appears in the action clause of a trigger, however, dropping the column can invalidate the trigger. The following statements illustrate the possible effects on triggers:

```
CREATE TABLE tab1 (i1 int, i2 int, i3 int);
CREATE TABLE tab2 (i4 int, i5 int);
CREATE TRIGGER col1trig UPDATE OF i2 ON tab1
   BEFORE(INSERT INTO tab2 VALUES(1,1));
ALTER TABLE tab2 DROP i4;
```

After the ALTER TABLE statement, **tab2** has only one column. The **col1trig** trigger is invalidated because the action clause as it is currently defined with values for two columns cannot occur.

If you drop a column that occurs in the triggering column list of an UPDATE trigger, the database server drops the column from the triggering column list. If the column is the only member of the triggering column list, the database server drops the trigger from the table. For more information on triggering columns in an UPDATE trigger, see CREATE TRIGGER statement on page 378.

If a trigger is invalidated when you alter the underlying table, drop and then re-create the trigger.

How Dropping a Column Affects Views

When you drop a column from a table, the views based on that table remain unchanged. That is, the database server does not automatically drop the corresponding columns from associated views.

The view is not automatically dropped because ALTER TABLE can change the order of columns in a table by dropping a column and then adding a new column with the same name. In this case, views based on the altered table continue to work, but retain their original sequence of columns.

If a view is invalidated when you alter the underlying table, you must rebuild the view by using the DROP VIEW and CREATE VIEW statements.

LOCK MODE Clause

Use the LOCK MODE keywords to change the locking granularity of an existing table.

This syntax fragment is part of the ALTER TABLE statement on page 95.

LOCK MODE Clause

```
" LOCK MODE "
" ( { PAGE | ROW } ) "
```

The following table describes the locking-granularity options available.

Granularity	Effect
PAGE	Obtains and releases one lock on a whole page of rows If no system default locking granularity has been set by the **IFX_DEF_TABLE_LOCKMODE** environment variable or by the **DEF_TABLE_LOCKMODE** configuration parameter, this is the default. Page-level locking is especially useful when you know that the rows are grouped into pages in the same order that you are using to process all

the rows. For example, if you are processing the contents of a table in the same order as its cluster index, `PAGE` is usually more appropriate setting than than `ROW`.

ROW

Obtains and releases one lock per row

Row-level locking supports the highest level of concurrency. Only tables with row-level locking support the LAST COMMITTED feature, which can improve performance in the Committed Read and Dirty Read isolation levels when another session holds an exclusive lock on a row that you attempt to read. If you are using many rows at one time, however, the lock-management overhead of row-level locking can become significant. You can also exceed the maximum number of locks available, depending on the configuration of your database server.

The following statement changes the lock mode for the **customer** table to page-level locking:

```
ALTER TABLE customer LOCK MODE(PAGE);
```

The next example changes the lock mode for the **customer** table to row-level locking:

```
ALTER TABLE customer LOCK MODE(ROW);
```

Important:

The SET LOCK MODE statement of SQL has no effect on the locking granularity of tables. You can use that statement, however, to define how the database server resolves locking conflicts when one process tries to access a row or a table that has been locked by a concurrent process. For the syntax and semantics, see SET LOCK MODE statement on page 817.

The term *lock mode* also has a third meaning for smart large objects in JDBC contexts, when a row contains one or more smart large objects. Here the scope of a lock can be either only a subset the smart large object, or else it can lock the entire BLOB or CLOB.

Precedence and Default Behavior

The LOCK MODE setting in an ALTER TABLE statement takes precedence over the settings of the **IFX_DEF_TABLE_LOCKMODE** environment variable and the DEF_TABLE_LOCKMODE configuration parameter. For information about the **IFX_DEF_TABLE_LOCKMODE** environment variable, refer to the *HCL® Informix® Guide to SQL: Reference*. For information about the DEF_TABLE_LOCKMODE configuration parameter, refer to the *HCL® Informix® Administrator's Reference*.

Logging TYPE Options

Use the Logging TYPE options to specify that the table has particular characteristics that can improve various bulk operations on it.

This syntax fragment is part of the ALTER TABLE statement on page 95.

Logging TYPE Options

" TYPE "

" ({ STANDARD | RAW }) "

Here STANDARD, the default option of the CREATE TABLE statement, specifies a logged table, and RAW specifies an unlogged table.

A table can have any of the following logging characteristics.

Option	Effect
STANDARD	Logging table that allows rollback, recovery, and restoration from archives. This is the default. Use this type for recovery and constraints functionality on OLTP databases.
RAW	Nonlogging table that do not support primary key constraints, unique constraints, or referential constraints. RAW tables can have NOT NULL constraints and NULL constraints (but not on the same set of columns). They can be indexed and updated. Use this type for quickly loading data.

Warning: Use RAW tables for fast loading of data. It is recommended that you alter the logging type to STANDARD and perform a level-0 backup before you use the table in a transaction or modify the data within the table. If you must use a RAW table within a transaction, either set the isolation level to Repeatable Read or lock the table in exclusive mode to prevent concurrency problems.

The Logging TYPE option can convert a non-logging table, such as a RAW table, to a STANDARD table that supports transaction logging. If you use this feature, you should be aware that the database server does not check to see whether a level 0 archive has been performed on the table.

Operations on a RAW table are not logged and are not recoverable, so RAW tables are always at risk. When the database server converts a table that is not logged to a STANDARD table type, it is your responsibility to perform a level-0 backup before using the table in a transaction or modifying data in the table. Failure to do this might lead to recovery problems in the event of a server crash.

For more information on these logging types of tables, refer to your *HCL® Informix® Administrator's Guide*.

The Logging TYPE options have the following restrictions:

- You must perform a level-0 archive before the logging type of a table can be altered to STANDARD from any other logging type.
- The table cannot be a TEMP table, and you cannot change any of these types of tables to a TEMP table.

The following example changes a nonlogging table to a table that uses transaction logging:

```
ALTER TABLE tabnolog TYPE (STANDARD);
```

The following example changes a logging table to a nonlogging table.

```
ALTER TABLE tablog TYPE (RAW);
```

MODIFY Clause

Use the MODIFY clause to change the data type, length, or default value of a column, to add or remove the security label of a column, to allow or disallow NULL values in a column, or to reset the serial counter of a SERIAL, SERIAL8, or BIGSERIAL column.

This syntax fragment is part of the ALTER TABLE statement on page 95.

(explicit id unique_144_Connect_42_modc002) unique_144_Connect_42_modc002(explicit id unique_144_Connect_42_modc003)

unique_144_Connect_42_modc003(explicit id unique_144_Connect_42_modc004) unique_144_Connect_42_modc004(explicit id

unique_144_Connect_42_modc005) unique_144_Connect_42_modc005(explicit id unique_144_Connect_42_modc006) unique_144_Connect_42_modc006

MODIFY Clause

" **MODIFY** "

" (<Modify Column Clause>) "

Modify Column Clause

" *column* "

" <Data Type> [] "

" { | [[] <DEFAULT Clause> []] | [[] <Single-Column Constraint Format> []] } "

" [<Modify Column Security> []] "

Element	Description	Restrictions	Syntax
column	Column to modify	Must exist in table. Cannot be a collection or IDSSECURITYLABEL data type.	Identifier on page 1231

Usage

When you modify a column, all attributes previously associated with the column (that is, default value, single-column check constraint, or referential constraint) are dropped. When you want certain attributes of the column to persist, such as PRIMARY KEY, you must respecify those attributes in the same MODIFY clause.

For example, consider the **items** table in the **stores_demo** database, whose schema includes this original definition of the **quantity** column:

```
CREATE TABLE items (item_num INT . . . -- primary key of this table
                    order_num    . . . -- foreign key to orders table
                                 . . . -- two additional columns
                    quantity SMALLINT DEFAULT 1 NOT NULL,
                    total price MONEY(8) . . . );
```

If you are changing the data type of the existing **quantity** column to INT, but you want to keep the default value (in this case, `1`) and the NOT NULL constraint, you can issue this statement:

```
ALTER TABLE items MODIFY (quantity INT DEFAULT 1 NOT NULL);
```



Note: Both the `DEFAULT` and `NOT NULL` attributes are specified again in the MODIFY clause. If you omit those keywords, both attributes are dropped from the modified column, and the modified table would accept NULL values in new or updated data rows in which the **quantity** value is missing.

When you specify a PRIMARY KEY constraint in the MODIFY clause, the database server also silently creates a NOT NULL constraint on the same column, or on the same set of columns that makes up the primary key

When you change the data type of a column, the database server does not perform the modification in place. The next example changes a VARCHAR(15) column to an LVARCHAR(3072) column:

```
ALTER TABLE stock MODIFY (description LVARCHAR(3072));
```

When you modify a column that has column constraints that are associated with it, the following constraints are dropped:

- All single-column constraints are dropped.
- All referential constraints that reference the column are dropped.
- If the modified column is part of a multiple-column primary-key or unique constraint, all referential constraints that reference the multiple columns also are dropped.

For example, if you modify a column that has a unique constraint, the unique constraint is dropped. If this column was referenced by columns in other tables, those referential constraints are also dropped. In addition, if the column is part of a multiple-column primary-key or unique constraint, the multiple-column constraints are not dropped, but any referential constraints that are placed on the column by other tables are dropped.

For another example, suppose that a column is part of a multiple-column primary-key constraint. This primary key is referenced by foreign keys in two other tables. When this column is modified, the multiple-column primary-key constraint is not dropped, but the referential constraints that are placed on it by the two other tables are dropped.

Consider the table that this statement defines:

```
CREATE TABLE tab1(c1 INT, c2 INT);
```

To add the NOT NULL constraint, an ALTER TABLE MODIFY statement is required:

```
ALTER TABLE tab1 MODIFY (c1 INT NOT NULL);
```

You cannot add a NULL or a NOT NULL constraint with the ADD CONSTRAINT clause.

Restrictions on changing column data types

You cannot use the ALTER TABLE MODIFY statement to change the data type of a column to a COLLECTION type or to a ROW type.

If the schema of a table exactly matches the order of data types in the fields of an existing named ROW type, however, you can use the ALTER TABLE ADD TYPE statement to change that table into a typed table, as the topic ADD TYPE Clause on page 127 describes.

The IDSSECURITYLABEL column of a protected table cannot be altered to a different data type, nor can an existing column be altered to be of type IDSSECURITYLABEL.

In general, the database server does not validate the resulting column values when you change the data types of columns in tables that contain data. The database server does not validate the values when you convert a column from an INTEGER or SMALLINT data type to a SERIAL, SERIAL8, or BIGSERIAL column.

Altering BYTE and TEXT Column data types

You can use the MODIFY clause to change a BYTE column to a TEXT column, and vice versa. You can also use the MODIFY clause to change a BYTE column to a BLOB column and a TEXT column to a CLOB column.

Except for these operations, however, you cannot use the MODIFY clause to change a BYTE or TEXT column to any other type of column, nor to change any other type of column to a BYTE or TEXT column.

When you use this clause to change a BYTE column to a BLOB column, or to change a TEXT column to a CLOB column, you can also use the PUT Clause on page 149 of the ALTER TABLE statement to specify an sbspace and to define its characteristics for storing the BLOB or CLOB objects.

Altering the Next Serial Value

You can use the MODIFY clause to reset the next value of a SERIAL, BIGSERIAL, or SERIAL8 column. You cannot set the next value below the current maximum value in the column because that action can cause the database server to generate duplicate numbers. You can set the next value, however, to any value higher than the current maximum, which creates a gap in the series of values.

If the new serial value that you specify is less than the current maximum value in the serial column, the maximum value is not altered. If the maximum value is less than what you specify, the next serial number will be what you specify. The next serial value is not equivalent to one greater than the maximum serial value in the column in two situations:

- There are no rows in the table, and an initial serial value was specified when the table was created (or by a previous ALTER TABLE statement).
- There are rows in the table, but the next serial value was modified by a previous ALTER TABLE statement.

The following example sets the next serial value to 1000:

```
ALTER TABLE my_table MODIFY (serial_num SERIAL (1000));
```

As an alternative, you can use the INSERT statement to create a gap in the series of serial values in the column. For more information, see Inserting Values into Serial Columns on page 585.

Altering the Next Serial Value in a Typed Table

You can set the initial serial number or modify the next serial number for a ROW-type field with the MODIFY clause of the ALTER TABLE statement. (You cannot set the initial number for a serial field when you create a ROW data type.)

Suppose you have ROW types **parent**, **child1**, **child2**, and **child3**.

```
CREATE ROW TYPE parent (a int);
CREATE ROW TYPE child1 (s serial) UNDER parent;
CREATE ROW TYPE child2 (b float, s8 serial8) UNDER child1;
CREATE ROW TYPE child3 (d int) UNDER child2;
```

You then create corresponding typed tables:

```
CREATE TABLE OF TYPE parent;
CREATE TABLE OF TYPE child1 UNDER parent;
CREATE TABLE OF TYPE child2 UNDER child1;
CREATE TABLE OF TYPE child3 UNDER child2;
```

To change the next SERIAL and SERIAL8 numbers to 75, you can issue the following statement:

```
ALTER TABLE child3 MODIFY (s serial(75), s8 serial8(75));
```

When the ALTER TABLE statement executes, the database server updates corresponding serial columns in the **child1**, **child2**, and **child3** tables.

Altering character columns

You can use the MODIFY clause to change the declared length of an existing CHAR, LVARCHAR, NCHAR, NVARCHAR, or VARCHAR column.

Similarly, the MODIFY clause can change the data type of a character column to a non-character data type.

For modified columns that you declare as built-in character data types, explicit or default size specifications are interpreted in units of bytes, unless the database was created with the SQL_LOGICAL_CHAR configuration parameter set to enable logical character semantics in character type declarations. For more information about logical character semantics when the ALTER TABLE statement declares size specifications for character columns, see Logical Character Support in Character Columns on page 102. For more information about the SQL_LOGICAL_CHAR configuration parameter, see your *HCL® Informix® Administrator's Reference*. For additional information about multibyte locales and logical characters, see the *HCL® Informix® GLS User's Guide*.

In a database that was created as NLSCASE INSENSITIVE, changing a character column of type NCHAR or NVARCHAR into type CHAR, LVARCHAR, or VARCHAR causes the database server to process values in the modified column as case-sensitive.

Conversely, in the same case-insensitive database, changing a character column of type CHAR, LVARCHAR, or VARCHAR into type NCHAR or NVARCHAR results in case-insensitive processing of values in the modified column. (The data values are not changed, but variations in letter case are ignored in comparison and collation operations on those values.)

Altering the Structure of Tables

About this task

When you use the MODIFY clause, you can also alter the structure of other tables. If the modified column is referenced by other tables, those referential constraints are dropped. You must add those constraints to the referencing tables again, using the ALTER TABLE statement.

When you change the data type of an existing column, all data is converted to the new data type, including numbers to characters and characters to numbers (if the characters represent numbers). The following statement changes the data type of the **quantity** column:

```
ALTER TABLE items MODIFY (quantity CHAR(6));
```

When a primary-key or unique constraint exists, however, conversion takes place only if it does not violate the constraint. If a data type conversion would result in duplicate values (by changing FLOAT to SMALLFLOAT, for example, or by truncating CHAR values), the ALTER TABLE statement fails.

Modifying Tables for NULL Values

About this task

You can modify an existing column that formerly permitted NULLs to disallow NULLs, provided that the column contains no NULL values. To do this, specify MODIFY with the same column name and data type and the NOT NULL keywords. Those keywords create a NOT NULL constraint on the column.

You can modify an existing column that did not permit NULLs to permit NULLs. To do this, specify MODIFY with the column name and the existing data type, and omit the NOT NULL keywords. The omission of the NOT NULL keywords drops the NOT NULL constraint on the column. If a unique index exists on the column, you can remove it using the DROP INDEX statement.

An alternative method of permitting NULL values in an existing column that did not permit NULL values is to use the DROP CONSTRAINT clause to drop the NOT NULL constraint on the column.

When you define a PRIMARY KEY constraint, the database server also silently creates a NOT NULL constraint on the same column, or on the same set of columns that make up the primary key.

Adding a Constraint on a Non-Opaque Column

ALTER TABLE ... MODIFY operations that use the Single Column Constraint format to implicitly create an index on a non-opaque column also automatically calculate the distribution of the specified column. The distribution statistics are available to the query optimizer when it designs query plans for the table on which the constraint is defined:

- For columns on which the new constraint is implemented as a B-tree index, the recalculated column distribution statistics are equivalent to distributions created by the UPDATE STATISTICS statement in HIGH mode.
- If the new constraint is not implemented as a B-tree index, the automatically recalculated statistics correspond to distributions created by the UPDATE STATISTICS statement in LOW mode.

See also the section Automatic Calculation of Distribution Statistics on page 261 in the description of the CREATE INDEX statement for additional information about statistical distributions that are produced automatically when an index or constraint is created on an existing table.

Modify Column Security clause

The DBSECADM can use this clause of the ALTER TABLE statement to modify the column-level security of a column, either by associating the column with an LBAC security label, or by dropping an existing association between the column and a security label.

This clause is valid only for tables that are protected by an LBAC security policy. This syntax fragment is part of the MODIFY Clause on page 139.

Modify Column Security Clause

" { [`COLUMN`] `SECURED WITH` *label* | `DROP COLUMN SECURITY` } "

Element	Description	Restrictions	Syntax
label	Name of an LBAC security label	Must exist and must belong to the same LBAC security policy that protects the table.	Identifier on page 1231

Usage

The DBSECADM can use this clause to add or drop column-level protection for a table. This clause can add row-level protection by associating the column with an LBAC security label, or it can use the `DROP COLUMN SECURITY` keywords to drop existing column-level security protection that a label was currently providing for the column.

- To establish column-level protection to a column, specify `SECURED WITH` *label* (or equivalently, `COLUMN SECURED WITH` *label*) to associate a security label with the column.
- To drop the column-level protection that an existing label is providing for a column, specify the `DROP COLUMN SECURITY` keywords.

The security label can be the same label that protects other rows or columns of the table, or it can be a different label of the same security policy. The following restrictions apply to the SECURED WITH *label* option:

- The table must be protected by an LBAC security policy.
- The data type of the protected column cannot be IDSSECURITYLABEL.
- The *label* must be a label of the same security policy that secures the table.
- Any number of columns can be protected by a security label, but no more than one security label can protect a given column.
- No more than one column name can precede each `SECURED WITH` or `DROP COLUMN SECURITY` specification of the MODIFY Clause on page 139,
- Specify the *label* without the policy qualifier, rather than as *policy*.*label*.

When the ALTER TABLE MODIFY COLUMN SECURED WITH statement or the ALTER TABLE MODIFY DROP COLUMN SECURITY statement run successfully, the database server registers the change in column security in the **syscolumns** system catalog table.

For example, if the table **Argoknot** is protected by a security policy called **Medea** that has a security label called **fleece**, the following ALTER TABLE statement defines column-level protection with the **Medea.fleece** label for the column **Argoknot.ColJ**:

```
ALTER TABLE Argoknot
   MODIFY (ColJ JSON SECURED WITH fleece);
```

When DBSECADM successfully runs this DDL statement, the new column-level security of column **Argoknot.ColJ** is registered in the **syscolumns** system catalog table. To do this, the database server takes this action:

- It updates the row in **syscolumns** that describes column **Argoknot.ColJ**
- by replacing the `NULL` value in column **syscolumns.seclabelid**
- with the numeric value in column **sysseclabels.seclabelid**
- for the row describing the **Medea.fleece** security label.

Similarly, DBSECADM can later detach the **Medea.fleece** security label from the **Argoknot.ColJ** column by running the following ALTER TABLE statement:

```
ALTER TABLE Argoknot
   MODIFY (ColJ JSON DROP COLUMN SECURITY);
```

After this DDL statement executes successfully, the database server registers the now unprotected status of column **ColJ** by updating column **seclabelid** to a NULL value in the row of the **syscolumns** system catalog table that describes column **Argoknot.ColJ**.

Example of enabling column-level security

Suppose that table **Alphanumeric** was created with the following columns but with no security policy:

```
CREATE TABLE Alphanumeric
   (ColB CHAR(18),
    Col2 INT,
```

```
    Col3 CHAR (20));
```

The following ALTER TABLE statement

- attaches the **watchdog** security policy to the **Alphanumeric** table,
- and changes the column-level security status of column **Col1** from *unprotected* to *protected*, by using the `SECURED WITH` keywords to associate that column with the LBAC **watchdog.Label23** security label:

```
ALTER TABLE Alphanumeric
   MODIFY (Col1 CHAR(18) SECURED WITH Label23)
   ADD SECURITY POLICY watchdog;
```

Because the **Alphanumeric** table was created with no security policy, the ADD SECURITY POLICY clause is required for the Modify Column Security clause to be valid in the example above.

The result of the ALTER TABLE MODIFY operation would be the same if the `COLUMN` keyword had preceded the `SECURED WITH` keywords:

```
ALTER TABLE Alphanumeric
   MODIFY (Col1 CHAR(18) COLUMN SECURED WITH Label23)
   ADD SECURITY POLICY watchdog;
```

In either case, the **Alphanumeric** table now has column-level protection for column **Col1**, which is secured by the **Label23** security label of the **watchdog** security policy.

If the data in other columns of the **Alphanumeric** table besides **Col1** have the same security requirements as **Col1**, the following statements secure the other columns with the same **Label23** security label:

```
ALTER TABLE Alphanumeric
    MODIFY (Col2 INT COLUMN SECURED WITH Label23);

ALTER TABLE Alphanumeric
    MODIFY (Col3 CHAR(20) COLUMN SECURED WITH Label23);
```

DBSECADM can also grant that security label to the specific users who require access to the data that **Label23** is designed to protect:

```
 GRANT SECURITY LABEL watchdog.Label23
   TO sam FOR READ ACCESS;

 GRANT SECURITY LABEL watchdog.Label23
   TO peter, elaine, olan FOR WRITE ACCESS;

 GRANT SECURITY LABEL watchdog.Label23
   TO lynette FOR ALL ACCESS;
```

For detailed examples of creating LBAC security labels and granting them to users, see Examples of Granting User Security Labels on page 565.

Dropping security-label protection from a column

Suppose table **Betanumeric** is a protected table that was created with **watchdog** as its security policy,

- with both column **Col1** and **Col3** secured by the column-level protection of label **labelK9**,
- and with column **Col4** providing row-level security for the table with a security label whose default value is **labelK2:**,

```
CREATE TABLE Betanumeric
   (Col1 CHAR (18)SECURED WITH labelK9,
    Col2 INT,
    Col3 CHAR (20)SECURED WITH labelK9
    Col4 IDSSECURITYLABEL DEFAULT 'labelK2')
   SECURITY POLICY watchdog;
```

Here **labelK9** and **1abelK2** must both be labels of the **watchdog** security policy.

Tip: For two or more columns of the same table to have column-level protection, it is not required that the same LBAC label secure all of those columns. If sensitive data in several columns logically require the same access restrictions, however, securing multiple columns with the same security label might be appropriate.

Suppose that DBSECADAM subsequently determines that the sensitivity of the data stored in column **Col1** does not require the protection of **watchdog.labelK9**, but is sufficiently protected by the row-level protection of label **watchdog.labelK2**. In this case, DBSECADAM might issue the following ALTER TABLE MODIFY statement, thereby dropping the association between security label **watchdog.labelK9** and **Col1** of table **Betanumeric**:

```
ALTER TABLE Betanumeric
   MODIFY (Col1 CHAR (18) DROP COLUMN SECURITY);
```

This ALTER TABLE MODIFY statement drops the security restriction that users cannot access column **Col1** unless their LBAC credentials include all the components of label **watchdog.labelK9**, or unless they hold equivalent LBAC security exemptions. Detaching the specified column from the specified security label is the entire scope of the `DROP COLUMN SECURITY` option to the Modify Column Security clause.

That is, disable disabling column-level security of a single column has no effect on the column-level protection of any other columns of the same table that are secured by the same LBAC label, such as column **Betanumeric.Col3** in this example. This `DROP COLUMN SECURITY` example also has no effect on any the following:

- On the row-level protection of table **Betanumeric** by the **watchdog.labelK2** label.
- On any other tables protected by the **watchdog** security policy,
- On any other tables protected by the **labelK9** security label.

Thus, only the column-level protection of column **Betanumeric.Col1** was dropped by the preceding ALTER TABLE MODIFY statement. The **watchdog** security policy, through row-level protection by the **labelK2** security label in column **Col4**, and column-level protection with the **labelK9** security label for column **Col3**, continues to protect table **Betanumeric**.

Detaching a table from its security policy

The DROP SECURITY POLICY clause detaches only a single column from its security label. If you hold the DBSECADM role, it is important to avoid confusing the DROP COLUMN SECURITY clause with the DROP SECURITY POLICY clause.

The following example, based on the same **Betanumeric** table, illustrates the ALTER TABLE statement syntax to drop the association between a specific table and its LBAC security policy:

```
ALTER TABLE Betanumeric DROP SECURITY POLICY;
```

When this statement executes successfully, it removes all column-level and row-level protection from the table. The database server updates the system catalog to show these changes to database objects within the scope of this example:

- Table **Betanumeric** is no longer protected by any security policy. This change is registered in two columns of the row that describes **Betanumeric** in the **systables** system catalog table:
  - In column **secpolicyid**, a `NULL` value replaces the numeric value of the **watchdog** security policy ID.
  - In column **protgranularity** of the same row, a blank character (ASCII 32) to indicate no protection replaces the `B` that had encoded both row-level and column-level protection as the former LBAC granularity of the **Betanumeric** table.
- Columns **Betanumeric.Col1** and **Betanumeric.Col3** are no longer protected by LBAC label **watchdog.labelK9**. This change is registered in two rows that describe those columns in the **syscolumns** system catalog table:
  - In column **seclabelid** of the row describing the **Betanumeric.Col1** column, a `NULL` value replaces the numeric value of the **labelK9** security label ID.
  - In column **seclabelid** of the row describing the **Betanumeric.Col3** column, a `NULL` value replaces the numeric value of the **labelK9** security label ID.
- Column **Col4** is automatically dropped from the schema of table **Betanumeric**, because a table with no security policy cannot include a column of type IDSSECURITYLABEL. This change is registered in the system catalog by dropping the row that had described the **Betanumeric.Col4** column from the **syscolumns** system catalog table.

Important: Except for the **Betanumeric** table and the data in its rows and columns, the ALTER TABLE DROP SECURITY POLICY statement in this example has no effect on any other database objects that the **watchdog** security policy or the **labelK2** or **labelK9** security labels protect. Only DROP SECURITY statements can destroy security objects. For more information about detaching a security policy from a table, see SECURITY POLICY Clause on page 154.

Adding a Constraint That Existing Rows Violate

If you use the MODIFY clause to add a constraint in the enabled mode and receive an error message because existing rows would violate the constraint, take the following steps to add the constraint successfully:

1. Add the constraint in the disabled mode.

   Issue the ALTER TABLE statement again, but this time specify the DISABLED keyword in the MODIFY clause.
2. Start a violations and diagnostics table for the target table with the START VIOLATIONS TABLE statement.
3. Issue the SET CONSTRAINTS statement to switch the database object mode of the constraint to the enabled mode.

   When you issue this statement, existing rows in the target table that violate the constraint are duplicated in the violations table; however, you receive an integrity-violation error message, and the constraint remains disabled.
4. Issue a SELECT statement on the violations table to retrieve the nonconforming rows that are duplicated from the target table.

You might need to join the violations and diagnostics tables to get all the necessary information.

5. Take corrective action on the rows in the target table that violate the constraint.
6. After you fix all the nonconforming rows in the target table, issue the SET statement again to enable the constraint that was disabled.

   Now the constraint is enabled, and no integrity-violation error message is returned because all rows in the target table now satisfy the new constraint.

How Modifying a Column Affects Triggers

If you modify a column that appears in the triggering column list of an UPDATE trigger, the trigger is unchanged.

When you modify a column in a table, the triggers based on that table remain unchanged, but the column modification might invalidate the trigger.

The following statements illustrate the possible affects on triggers:

```
CREATE TABLE tab1 (i1 INT, i2 INT, i3 INT);
CREATE TABLE tab2 (i4 INT, i5 INT);
CREATE TRIGGER col1trig UPDATE OF i2 ON tab1
   BEFORE(INSERT INTO tab2 VALUES(1,1));
ALTER TABLE tab2 MODIFY i4 CHAR;
```

After the ALTER TABLE statement, column **i4** accepts only character values. Because character columns accept only values enclosed in quotation marks, the action clause of the **col1trig** trigger is invalidated.

If a trigger is invalidated when you modify the underlying table, drop and then re-create the trigger.

How Modifying a Column Affects Views

When you modify a column in a table, the views based on that table remain unchanged. If a view is invalidated when you alter the underlying table, you must rebuild the view.

PUT Clause

Use the PUT clause of the ALTER TABLE statement to define sbspace storage for a column that contains smart large objects. This clause can define BLOB or CLOB storage attributes for a new column, or it can modify the current settings of those storage attributes for an existing column.

This syntax fragment is part of the ALTER TABLE statement on page 95.

PUT Clause

" PUT *column* "

" IN (*sbspace*) "

" [([{ | EXTENT SIZE *kilobytes* | [{ NO LOG | LOG }] | { HIGH INTEG | [MODERATE INTEG] } | [{ NO KEEP ACCESS TIME | KEEP ACCESS TIME }] }])] "

Element	Description	Restrictions	Syntax
column	Column to store in the specified sbspace	Must be a UDT, or a complex, BLOB, or CLOB data type	Identifier on page 1231
kilobytes	Number of kilobytes to allocate for the extent size	Must be an integer value	Literal Number on page 1198
sbspace	Name of an area of storage for smart large objects	The sbspace must exist. See also **Usage** for *sbspace* restrictions on tables in tenant databases.	Identifier on page 1231

Usage

The data type of the *column* can be BLOB or CLOB, or it can be a complex data type or a user-defined data type (UDT) that can store smart large objects.

The format *column.field* is not valid here. That is, the smart large object that you are storing cannot be one field of a ROW type.

If the table that the PUT clause of the ALTER TABLE statement is modifying is in a tenant database, the *sbspace* must be a dedicated sbspace in the tenant database properties list. If the table is not in a tenant database, the *sbspace* cannot be the name of an sbspace that is dedicated to a tenant database.

Specifying more than one *sbspace* distributes the storage of the BLOB or CLOB objects in a round-robin distribution scheme. When a new row is inserted, the database server favors the sbspace with the most available storage space.

When more than one *sbspace* name follows the `IN` keyword, parentheses must delimit the comma-separated list of sbspaces that store the BLOB or CLOB objects. Unless you accept default values for all of the storage options that can follow the *sbspace* list, delimiting parentheses must also enclose the list of storage options, as in the following example:

```
ALTER TABLE MyClobs ADD (c5 CLOB)
   PUT c5 IN (sbs3,sbs4) (EXTENT SIZE 64, HIGH INTEG);
```

Here the ADD clause above appends a new column **c5** of type CLOB to table **MyClobs**. The PUT clause stores each CLOB object in sbspace **sbs3** or **sbs4**, in a round-robin distribution. The first extent size is 64 kilobytes, and the data pages will

include page headers and page footers in `HIGH` data integrity mode. By default, the column is unlogged, and access times are not recorded.

When the PUT clause modifies a BLOB or CLOB column, the storage characteristics of smart large objects already stored in the column are unchanged. The new characteristics apply only to BLOB or CLOB objects in rows inserted into the table after the PUT clause takes effect.

This syntax resembles the PUT clause of the CREATE TABLE statement, but the PUT clause of ALTER TABLE can specify only a single column, rather than a list of columns. Changes made to storage attributes of BLOB or CLOB objects by the PUT clauses of the ALTER TABLE statement or the CREATE TABLE statement are registered in the **syscolattribs** system catalog table.

Using the PUT clause with a new or modified column

When you use this clause to specify storage attributes for smart large objects in a new column that the ALTER TABLE statement is adding to the table schema, or for an existing column whose sbspace storage the ALTER TABLE statement is modifying, these keyword options of the PUT clause can define explicit or default characteristics for one or more existing sbspace:

`IN (` *sbspace_list*)

Here *sbspace_list* is a comma-separated list of one or more sbspace names. If you omit `IN` *sbspace_list* from the PUT clause, the **SBSPACENAME** configuration parameter specifies the system default sbspace in which smart large objects are stored. If the *sbspace_list* of the PUT clause specifies more than one sbspace, the smart blob objects are distributed round-robin among those sbspaces. Each smart large object is stored in a single sbspace.

`EXTENT SIZE` *integer*

This sets a lower limit on how many kilobytes can be stored in a smart-large-object extent. If you omit the `EXTENT SIZE` keywords, the database server calculates a default extent size.

`NO LOG`

This prevents the same logging procedure that is applied to other columns from being applied to the smart large objects. This default reduces log traffic, and reduces the risk of filling the logical log.

`LOG`

This alternative option to `NO LOG` applies the logging procedure used with the current database for smart large objects in the sbspaces that this PUT clause assigns.

`HIGH INTEG`

This high data-integrity option produces user-data pages that each contains a page header and a page trailer to detect incomplete writes and data corruption. This option is the default data-integrity behavior.

`MODERATE INTEG`

This alternative data-integrity option to `HIGH INTEG` produces user-data pages that each contains a page header but no page trailer. This option reduces operational costs, but cannot detect incomplete writes or data corruption by comparing the page header with a page trailer.

NO KEEP ACCESS TIME

This does not record the system time when the smart large object was last read or written. This option provides better performance than the `KEEP ACCESS TIME` option, and is the default tracking behavior.

KEEP ACCESS TIME

This alternative tracking option to `NO KEEP ACCESS TIME` maintains a record in the smart-large-object metadata of the system time when the smart large object was last read or written.

The PUT clause cannot specify a buffering mode or a locking granularity for the smart large object. The system default buffering mode is `OFF`, and the system default locking granularity is the entire smart large object. The -Df option of the onspaces utility can override these defaults for specific sbspaces that store BLOB or CLOB objects that require other settings.

When you modify the storage characteristics of an existing column, by default *all* attributes previously associated with the storage space for that column are dropped. If you want the same settings for certain attributes to persist, you must explicitly specify the settings for those attributes again. For example,

- to retain logging, you must specify the `LOG` keyword again. Otherwise, the default `NO LOG` option takes effect.
- to retain the less costly level of data-integrity, you must specify the `MODERATE INTEG` keywords again. Otherwise, the default `HIGH INTEG` option takes effect.
- to maintain a smart-large-object metadata record of when the smart large object was last read or written, you must specify `KEEP ACCESS TIME` keywords again. Otherwise, the default `NO KEEP ACCESS TIME` option takes effect.

Sections that follow illustrate ALTER TABLE operations that use these options of the PUT clause.

Suppose that permanent table **sbtab** was created with the following schema:

```
CREATE TABLE sbtab (c2 INT);
```

Example 1: Adding a new BLOB column to a table

The following ALTER TABLE statement alters the schema of table **sbtab** by adding BLOB column **c1** as the new first column, and stores its BLOB objects in existing sbspace **sbs1**, for which the PUT clause defines additional attributes:

```
ALTER TABLE sbtab ADD (c1 BLOB BEFORE c2)
   PUT c1 IN (sbs1) (EXTENT SIZE 32,
                     NO LOG,
                     MODERATE INTEG,
                     KEEP ACCESS TIME);
```

The PUT clause specifies these storage attributes for new column **c1**:

- allocates 32 kilobytes for the first extent of each BLOB object,
- disables transaction logging for the new column in sbspace **sbs1**,
- uses the data-integrity option of data page headers without footers,
- and maintains a record of when each BLOB object was last read or written.

Examples 2 and 3: Modifying sbspace attributes

Suppose that the owner of table **sbtab** subsequently decides to change the storage options for the BLOB objects in column **c1**. The following ALTER TABLE statement alters the storage attributes of BLOB column **c1** by changing the extent size in the sbspace **sbs1** to 64 kilobytes, and turning on transaction logging:

```
ALTER TABLE sbtab PUT c1 IN (sbs1) (EXTENT SIZE 64, LOG);
```

Because this example does not repeat the nondefault settings for the data integrity and access time attributes that *Example 1* specified, the new PUT clause sets these storage options for new BLOB objects in column **c1**:

- continues to store new BLOB objects in sbspace **sbs1**,
- increases to 64 kilobytes the first extent of each BLOB object,
- explicitly enables transaction logging for column **c1**,
- implicitly restores the default data-integrity option of data pages with headers and footers,
- and implicitly discontinues recording `ACCESS TIME` records.

But all of the storage attributes that the `ALTER TABLE sbtab ADD` statement set for sbspace **sbs1** in *Example 1* persist in any rows that were added to the **sbtab** table before *Example 2* reset those attributes.

The next example of the PUT clause modifies only one attribute of the same sbspace **sbs1** to change its logging mode for BLOB column **c1** to NO LOG:

```
ALTER TABLE sbtab PUT c1 IN (sbs1) (EXTENT SIZE 64, NO LOG);
```

Because NO LOG is the default for smart large objects, this statement has the same effect if the `NO LOG` keywords are omitted, as in this equivalent example:

```
ALTER TABLE sbtab PUT c1 IN (sbs1) (EXTENT SIZE 64);
```

Example 4: Distributing storage across sbspaces

This example alters the **sbtab** table to distribute the storage of BLOB column **c1** in sbspaces **sbs1** and **sbs2**.

```
ALTER TABLE sbtab PUT c1 IN (sbs1, sbs2)
   (EXTENT SIZE 100, LOG, KEEP ACCESS TIME);
```

The PUT clause also

- changes the extent size to 100 kilobytes,
- turns on transaction logging,
- and records the system times of the last read access and write access in the smart-large-object metadata.

For more information on the storage characteristics for smart large objects when database tables are created, refer to the counterpart of this topic in the PUT Clause on page 352 of the CREATE TABLE statement. For a discussion of large-object characteristics, refer to Large-Object Data Types on page 993.

SECURITY POLICY Clause

Use the Security Policy clause of the ALTER TABLE statement to drop the security policy that is currently associated with the table, or to associate a security policy with a table that has none.

This syntax fragment is part of the ALTER TABLE statement on page 95.

SECURITY POLICY Clause

"{ **ADD SECURITY POLICY***policy* **| DROP SECURITY POLICY** }"

Element	Description	Restrictions	Syntax
policy	Name of a security policy	Must be an existing security policy	Identifier on page 1231

Usage

Only DBSECADM can use this clause to add a security policy to an existing table, or to remove from the table the protection of the security policy that currently protects a table.

Restrictions on adding a security policy

The following guidelines apply to tables that can be protected by executing the ADD SECURITY POLICY clause of the ALTER TABLE statement:

- A table is not protected unless it has a security policy associated with it and has either rows secured, or has at least one column secured.
  - Having rows secured indicates that the table is a *protected table* with *row-level* granularity.
  - Having at least one column secured indicates that the table is a *protected table* with *column-level* granularity.
- Securing rows by using the ALTER TABLE ... ADD *column* statement to add an IDSSECURITYLABEL column to an existing table fails if the table does not have a security policy associated with it.
- Securing a column with the ALTER TABLE ... MODIFY *column* SECURED WITH *label* clause fails if the table does not have a security policy associated with it.
- A table can have at most one security policy. The ALTER TABLE ... ADD SECURITY POLICY statement fails if the table already has a security policy.
- A table can have any number of protected columns. Each protected column can have a different security label, or several protected columns can share the same security label, but all labels must have the same security policy.
- This clause cannot add a security policy to protect any of the following table objects
  - a temporary table,
  - a table outside the current database,
  - a typed table in a table hierarchy,
  - an object defined by the CREATE EXTERNAL TABLE statement.
- A table can have at most one column of type IDSSECURITYLABEL.

- The IDSSECURITYLABEL column cannot have column-level protection (that is, a security label).
- The IDSSECURITY LABEL column cannot have single column constraints.
- The IDSSECURITY LABEL column cannot be part of the multiple column key of a referential constraint or of a check constraint.
- The IDSSECURITYLABEL column cannot be encrypted.
- The IDSSECURITYLABEL column has an implicit DEFAULT NOT NULL constraint. The default column value is the value of the security label of the user for write access.
- The IDSSECURITYLABEL column cannot be modified by the ALTER TABLE MODIFY statement.

Security policies and tables with distributed storage

Detaching a fragment of a protected table creates a new table that is protected by the same security policy for the same row security label column, and the same set of protected columns.

Some restrictions apply to associating a fragmented table with a security policy. Attaching a fragment to a protected fragmented table fails if any of these conditions are true:

- if the source table and the target table are not protected using the same security policy;
- if the tables do not have the same protection granularity;
- if the tables do not have the same set of protected columns, each protected by the same security label.

For more information on using the ALTER FRAGMENT statement to attach fragments to protected tables, see Additional Restrictions on the ATTACH Clause on page 30.

Associating a security policy with a table

In a database in which the LBAC security policy **Watchdog** is defined, consider an unprotected table with the following schema:

```
CREATE TABLE IF NOT EXISTS MyData
   (C1 CHAR (8),
   C2 INT,
   C3 CHAR (10));
```

The DBSECADM can change this to a table with row-level protection of the **Watchdog** security policy by issuing the following ALTER TABLE statement:

```
ALTER TABLE MyData
   ADD C4 IDSSECURITYLABEL,
   ADD SECURITY POLICY Watchdog;
```

Here no security label is specified, so the default security label for the rows of the **MyData** table is the label of the **Watchdog** security policy that the current user holds for write access. This example illustrates a requirement that if the table is protected by no security policy, both the ADD COLUMN IDSSECURITYLABEL clause and the ADD SECURITY POLICY clause must be included in the same ALTER TABLE statement.

An alternative to the syntax illustrated by the previous example is that the DBSECADM can instead change the **MyData** table to a table protected by a specific security label of the **Watchdog** security policy, rather than by a default security label, as in this ALTER TABLE statement:

```
ALTER TABLE MyData
   ADD C4 IDSSECURITYLABEL DEFAULT 'canine',
   ADD SECURITY POLICY Watchdog;
```

Here the security label **canine** must be part of the **Watchdog** security policy.

Dropping a security policy from a table

Conversely, in a database in which the LBAC security policy **Watchdog** is defined, consider a protected table called **MyOtherData** with the same schema that resulted after the ALTER TABLE statement in the previous example modified for the **MyData** table:

```
CREATE TABLE IF NOT EXISTS MyOtherData
   (C1 CHAR (8),
   C2 INT,
   C3 CHAR (10)
   C4 IDSSECURITYLABEL DEFAULT canine NOT NULL)
   SECURITY POLICY Watchdog;
```

The DBSECADM can change this **MyOtherData** table to an unprotected table with no security policy by issuing the following statement:

```
ALTER TABLE MyOtherData
   DROP SECURITY POLICY Watchdog;
```

When a security policy is dropped from a table by the ALTER TABLE DROP SECURITY POLICY statement, the IDSSECURITYLABEL column **C4** that provided row-level protection is automatically dropped. If labels table had protected any columns, those columns become unprotected when the security policy was dropped.

The DROP SECURITY POLICY option requires the Connect, Resource, and Alter discretional access privileges for dropping columns, because this option drops the IDSSECURITYLABEL column that only tables associated with a security policy can include in their schema.

Replacing the security policy of an existing table

If the DROP SECURITY POLICY clause of the ALTER TABLE statement executes successfully, the table is no longer protected by any security policy. Other tables that are associated with the same security policy, however, are not affected by the alterations to this table. The IDSSECURITYLABEL column and any security label associated with the dropped security policy are no longer part of the table schema.

If no protection of a security policy is needed for the data in the table, no further action is needed.

If either row-level or column-level protection is needed for the table, however, a user who holds DBSECADM credentials should take steps to attach another security policy to the table :

- Replace the security policy that was dropped with another security policy, using the ADD SECURITY POLICY clause in an ALTER TABLE statement that also establishes row-level or column-level security. The following example establishes both row-level and column-level protection for the **MyOtherData** table, based on the **Robodog** security policy:

```
ALTER TABLE MyOtherData
   ADD (C4 IDSSECURITYLABEL DEFAULT),
   MODIFY (C2 INT COLUMN SECURED WITH label7),
   ADD SECURITY POLICY Robodog;
```

- If the new security design requires only row-level protection, then instead of the previous example, the **C2** column need not be associated with **label7**:

```
ALTER TABLE MyOtherData
   ADD (C4 IDSSECURITYLABEL DEFAULT),
   ADD SECURITY POLICY Robodog;
```

- If the new security design requires only column-level protection, use the MODIFY . . . COLUMN SECURED WITH clause to protect one of more columns with labels of the new security policy.

```
ALTER TABLE MyOtherData
   MODIFY (C1 CHAR (8) COLUMN SECURED WITH label9),
   ADD SECURITY POLICY Robodog;
```

Dropping a security policy from a table or from the database

Do not confuse this DROP SECURITY POLICY clause of the ALTER table statement with the DROP SECURITY POLICY statement of SQL.

- When the DROP SECURITY POLICY clause of the ALTER TABLE statement executes successfully, its immediate effects include these:
  - It terminates the association of that table with the security policy,
  - and it drops the IDSSECURITYLABEL column from that table,
  - and it removes LBAC protection from data that had been protected in that table.

  It has no effect, however, on the security policy, nor on other tables protected by the policy.
- When the DROP SECURITY POLICY statement executes successfully, the scope of its effects depends on whether the policy is dropped in `RESTRICT` mode or in `CASCADE` mode, but in either mode, it destroys the specified policy.

  See the description of DROP SECURITY statement on page 485 for more information on the DROP SECURITY POLICY statement of SQL, and about restrictions on that statement.

Statistics options of the ALTER TABLE statement

Use the Statistics Options clause of the ALTER TABLE statement to change values of the STATCHANGE property of a fragmented or nonfragmented table, and the STATLEVEL property of a fragmented table. These table properties control the threshold for recalculation in automatic mode, and the granularity of data distribution statistics.

This clause supports the same syntax options as the Statistics Options clause of the CREATE TABLE statement. This syntax fragment is part of the ALTER TABLE statement on page 95.

Statistics Options

" [STATCHANGE { AUTO | *change_threshold* }] "

" [STATLEVEL { FRAGMENT | TABLE | AUTO }] "

Element	Description	Restrictions	Syntax
change_threshold	Percentage of changed data rows that defines stale distribution statistics	Must be an integer in the range 0 - 100	Literal Number on page 1198

Usage

The Statistics Options clause of the ALTER TABLE statement can define or modify table statistics properties that can affect these operations while the automatic mode of UPDATE STATISTICS is enabled:

- UPDATE STATISTICS statements without the FOR keyword
- UPDATE STATISTICS FOR TABLE statements in LOW, MEDIUM, or HIGH mode.

The ALTER TABLE statement can modify the specified or default values of the STATCHANGE and STATLEVEL properties that were set when the table was created, or that were set by a previous ALTER TABLE statement.

The STATCHANGE property

The STATCHANGE table property specifies the minimum percentage of changes (from UPDATE, DELETE, LOAD, and INSERT operations) on the rows in the table or fragment since the previous calculation of distribution statistics) to consider the statistics stale. You can specify the percentage change as either an integer value in the range 0 - 100, or you can use the AUTO keyword to apply the current STATCHANGE configuration parameter setting in the ONCONFIG file, or the setting in the session environment, as the default change threshold value.

The AUTO keyword of the UPDATE STATISTICS statement also enables comparing the proportion of rows with changed values to the STATCHANGE setting to determine whether the statistics in the system catalog are stale. Including the AUTO keyword in the UPDATE STATISTICS statement enables checking for stale statistics (and selectively updating only the tables or fragments with stale or missing statistics) during the current UPDATE STATISTICS operation.

When the AUTO_STAT_MODE configuration parameter or the AUTO_STAT_MODE session environment variable enables the automatic mode, the UPDATE STATISTICS statement uses the explicit or default STATCHANGE value to identify table, index, or fragment distributions whose statistics are missing or stale, and updates only the missing or stale statistics. For more information about the automatic mode for UPDATE STATISTICS operations, see information about the AUTO_STAT_MODE configuration parameter in the *HCL® Informix® Administrator's Reference*.

The STATCHANGE property and the automatic mode of UPDATE STATISTICS do not directly affect the optimization of SPL execution plans, or UPDATE STATISTICS statements that include the FORCE, FOR FUNCTION, FOR PROCEDURE, FOR ROUTINE, or FOR SPECIFIC keywords.

The STATLEVEL property

The STATLEVEL property can determine the level of granularity of the data distributions and index statistics of fragmented tables. It can take one of the following three values, with `AUTO` being the default, if no value is specified at creation time:

- `TABLE` specifies that all distributions for the table be created at the table level.
- `FRAGMENT` specifies that distributions be created and maintained for each fragment.
- `AUTO` specifies that the database server apply criteria at run time to determine whether fragment-level distributions are necessary. These criteria require that the following conditions are true:
  - The SYSSBSPACENAME configuration parameter setting specifies an existing sbspace
  - The table is fragmented by an EXPRESSION, INTERVAL, Rolling Window, or LIST strategy
  - The table has more than a million rows.

  If any of these criteria are not satisfied, the database server creates table-level distributions, rather than fragment-level.

These properties are always applied. If the STATLEVEL setting is `AUTO`, this setting overrides the default values.

Note: The SYSSBSPACENAME configuration parameter, which must be set when the database server instance is initialized, specifies the sbspace in which the database server stores fragment-level data distribution statistics. These are stored as BLOB objects in the **encdist** column of the **syfragsdist** system catalog table. For the database server to support fragment level statistics, the SYSSBSPACENAME configuration parameter setting must specify an existing sbspace.

If you use the Statistics Options clause to set the STATLEVEL property to FRAGMENT, the database server returns an error -9814 ("`Invalid default sbspace name`") if either of the following is true:

- The SYSSBSPACENAME configuration parameter is not set
- The sbspace that SYSSBSPACENAME specifies was not properly allocated by the onspaces -c -S command.

Example of changing the STATLEVEL

Suppose that table **tabFrag** uses a fragmented distribution strategy other than ROUND ROBIN, and includes a BLOB or CLOB column called **smartblob**. You decide to keep the storage distribution strategy, but to use TABLE, rather than FRAGMENT, as the STATLEVEL granularity.

The following SQL statements implement those changes:

```
ALTER TABLE tabFrag STATLEVEL TABLE;

UPDATE STATISTICS LOW
   FOR TABLE tabFrag (smartblob) DROP DISTRIBUTIONS;

UPDATE STATISTICS HIGH
   FOR TABLE tabFrag (smartblob);
```

The statements above have these respective effects:

- Change the STATLEVEL to TABLE, by using the Statistics Options clause of ALTER TABLE.
- Discard the current fragment-level distributions of **tabFrag.smartblob** in the **sysfragdist** system catalog table, by using UPDATE STATISTICS LOW.
- Create new table-level statistics for **tabFrag** in the **sysdistrib** system catalog table, by using UPDATE STATISTICS HIGH.

In the last statement above, the default HIGH resolution of `0.5` implies that the **tabFrag.smartblob** distribution statistics are based on approximately 200 bins.

Example of resetting the STATCHANGE value

For the same **tabFrag** table whose STATLEVEL property was changed from FRAGMENT to TABLE in the previous section, suppose that you also decide to change its STATCHANGE value to AUTO from whatever setting it currently has, and to replace the HIGH mode distribution statistics with a MEDIUM mode.

The following SQL statements implement those changes:

```
ALTER TABLE tabFrag STATCHANGE AUTO;

UPDATE STATISTICS LOW
   FOR TABLE tabFrag (smartblob) DROP DISTRIBUTIONS;

UPDATE STATISTICS MEDIUM
   FOR TABLE tabFrag (smartblob) AUTO;
```

In the last s

These are the respective effects on the **tabFrag** table of the statements above:

- The Statistics Options clause of ALTER TABLE changes the STATCHANGE setting to AUTO.
- The UPDATE STATISTICS LOW statement discards the current fragment-level distributions of **tabFrag.smartblob** from the **sysfragdist** system catalog table, but recalculates no table-level statistics. (Here the `LOW` keyword is required syntax to enable the `DROP DISTRIBUTIONS` operation.)
- The UPDATE STATISTICS MEDIUM statement refresh the table-level statistics for **tabFrag** in the **sysdistrib** system catalog table in automatic mode, by including the `AUTO` keyword.

tatement above, the default MEDIUM resolution of `2.5` implies that the **tabFrag.smartblob** distribution statistics are based on approximately 40 bins.

In this example, the `AUTO` setting for STATCHANGE in the ALTER TABLE statement and the `AUTO` keyword in the UPDATE STATISTICS MEDIUM apply automatic mode to the recalculation of **tabFrag.smartblob** statistics. Although no **tabFrag.smartblob** data values changed since the UPDATE STATISTICS HIGH operation in the STATLEVEL example, the preceding UPDATE STATISTICS LOW statement dropped the HIGH mode statistics, so the statistics for 100% of the rows became "missing" during the DROP DISTRIBUTIONS operation. In this case, no selective recalculation of **tabFrag.smartblob** statistics occurs, despite the automatic mode.

The examples above illustrate some effects of ALTER TABLE options for redefining the statistics properties of a table, and the effects of those modified properties on some UPDATE STATISTICS operations in automatic mode. They do not

necessarily illustrate a recommended or efficient sequence of decisions for redefining the granularity or the resolution of column statistics for tables and for table fragments in the system catalog.

ALTER TRUSTED CONTEXT statement

Use the ALTER TRUSTED CONTEXT statement to modify the current options and attributes (including the ENABLED or DISABLED mode) of a trusted-context object.

This statement is an extension to the ANSI/ISO standard for the SQL language.

Syntax

(explicit id unique_208_Connect_42_alttc01) unique_208_Connect_42_alttc01 **ALTER TRUSTED CONTEXT***context* { | **ALTER** { [**SYSTEM AUTHID***user* | [**ATTRIBUTES(** **ADDRESS***address*) | [{ **NO DEFAULT ROLE** | **DEFAULT ROLE***role* } | [{ **DISABLE** | **ENABLE** } } | [**ADD ATTRIBUTES(** **ADDRESS***address*) | [**DROP ATTRIBUTES(** **ADDRESS***address*) | [**ADD USE FOR** <Authorized User Clause> | [**REPLACE USE FOR** <Authorized User Clause> | [**DROP USE FOR** { *user* | **PUBLIC** } }

Authorized User Clause

" { *user* [**ROLE***role*] | **PUBLIC** } [{ **WITHOUT AUTHENTICATION** | **WITH AUTHENTICATION** }] "

Element	Description	Restrictions	Syntax
address	Communication address of the client connection to the database server	Must be unique among communication addresses of clients for this trusted-context object. For additional *address* restrictions, see **ADDRESS attributes** below.	Quoted String on page 1202
context	Name of the trusted-context object	Must exist among trusted contexts of the database server instance, and cannot begin with the characters SYS	Identifier on page 1231
role	An existing user-defined or built-in role	Must exist in the database, and must be unique among attributes of this trusted-context object	Owner name on page 1262
user	Authorization identifier of a user	Must be a valid authorization identifier. Cannot be longer than 32 bytes. Must not be the authorization ID of the user who issues this statement. The REPLACE USE FOR clause must not specify this *user* more than once.	Owner name on page 1262

Usage

You must hold the database security administrator (DBSECADM) role to run this statement. If the statement is embedded in an application program, the privileges are those of the owner. If the statement is run in a trusted context with a role, the set of privileges is the union of these discretionary access privileges:

- the set of privileges held by the role that is associated with the primary authorization ID,
- and the set of privileges held by each role that the statement references.

When the ALTER TRUSTED CONTEXT statement executes successfully, any changes to the trusted-context object, its attributes, and its list of authorized users are registered in these tables in the sysuser database of the Informix® database server instance:

- **systrustedcontext**
- **systcxattributes**
- **systcxusers**.

ADDRESS attributes

The ALTER ATTRIBUTES, ADD ATTRIBUTES, and DROP ATTRIBUTES options can specify lists of one or more communication addresses for connections to the database server, whose status as connection trust attributes, on which the trusted-context object is defined, are to be modified. The following restrictions apply to communication addresses that the ALTER TRUSTED CONTEXT or the CREATE TRUSTED CONTEXT statements reference:

- Each must be unique among communication addresses of clients for this trusted-context object.
- Each must conform to the TCP/IP protocol.
- Each must be an IPv4 address, an IPv6 address, or a secure domain name.
- An IPv4 address or IPv6 address must be a real host address (not a local host), and must not contain leading blank spaces.
- An IPv6 address, in addition, must not be an IPv4-mapped IPv6 address.
- A secure domain name must not be a Dynamic Host Configuration Protocol (DHCP) address.

If a new *address* value is the name of a secure domain, that name is converted to an IP address by the domain-name server, where a resulting IPv4 or IPv6 address is determined. When a domain name is converted to an IP address, the result of this conversion might be a set of one or more IP addresses. In this case, the database server interprets a subsequent incoming connection request as matching the ADDRESS attribute of a trusted-context object if the IP address from which the connection originates matches any of the IP addresses to which the domain name was converted.

The ALTER ATTRIBUTES clause replaces existing values for specified attributes with the new values. If an attribute is not currently part of the trusted-context object definition, an error is returned. Attributes that are not specified retain their previous values.

Specified ADDRESS values for the trusted-context object are removed by the ALTER ATTRIBUTESclause and by the DROP ATTRIBUTES clause. The ADDRESS attribute can be specified multiple times, but each *address* value must be unique for the set of attributes.

If a new *address* value is the name of a secure domain, that name is converted to an IP address by the domain-name server, where a resulting IPv4 or IPv6 address is determined. When a domain name is converted to an IP address, the result of this conversion might be a set of one or more IP addresses. In this case, the database server interprets a subsequent incoming connection request as matching the ADDRESS attribute of a trusted-context object if the IP address from which the connection originates matches any of the IP addresses to which the domain name was converted.

The ADD ATTRIBUTES clause specifies a list of one or more new trust attributes on which the trusted-context object is defined.

The DROP ATTRIBUTES clause specifies that one or more attributes are to be dropped from the definition of the trusted-context object. If the attribute is not currently part of the trusted-context object definition, an error is returned.

Attention:

If you have an existing application with an ALTER TRUSTED CONTEXT statement that includes the `ENCRYPTION` or `WITH ENCRYPTION` options in the ATTRIBUTES clause, the database server does not issue an SQL error. Except, however, for the `WITH ENCRYPTION 'NONE'` and the `ENCRYPTION 'NONE'` keyword options, the encryption options of the ALTER TRUSTED CONTEXT statement are not supported for Informix® database servers.

DEFAULT ROLE attributes

The `DEFAULT ROLE` *`role`* option to the ALTER clause identifies a role that already exists on the current database server. This role can be used by a user who does not hold a user-specific role that was defined as part of the definition of the trusted-context object.

The `NO DEFAULT ROLE` keywords specify that the trusted-context object does not have a default role.

If a trusted connection for this trusted context is active, changes to the DEFAULT ROLE attribute take effect on the next new connection request, or on the next request to switch users.

ENABLE and DISABLE options of the ALTER clause

The ENABLE attribute specifies that the trusted-context object is enabled.

The DISABLE attribute specifies that the trusted-context object is in a disabled state, and is not enabled for any new trusted connections that are established.

You cannot use the SET Database Object Mode statement of SQL to change the ENABLE or DISABLE attributes of trusted contexts.

ADD USE FOR clause

The ADD USE FOR clause specifies additional users who can establish a trusted connection based on this trusted-context object. The PUBLIC attribute specifies that a trusted connection that is based on this trusted-context object can be used by any user.

The PUBLIC attribute must not already be specified for the trusted-context object, and PUBLIC must not be specified more than once in the ADD USE FOR clause. If the definition of a trusted-context object allows access by PUBLIC and also by one or more users, the user specifications override the PUBLIC specifications.

REPLACE USE FOR clause

The REPLACE USE FOR clause specifies that the way in which a specified user or the PUBLIC group uses the trusted-context object is to change. When you use the REPLACE USE FOR clause on PUBLIC, the trusted-context object must already be defined to allow use by PUBLIC, and PUBLIC must not be specified more than once in the REPLACE USE FOR clause.

The REPLACE USE FOR can specify a different role name, which must be a role that is defined on the database server. The role explicitly specified for the user overrides the default role, if a default role is currently associated with the trusted context.

The REPLACE USE FOR clause can also change the current authentication requirement for a user or for the PUBLIC group.

AUTHENTICATION attributes

The REPLACE USE FOR and ADD USE FOR clause can specify the authentication requirement for trusted connections based on this trusted-context object. The default is WITHOUT AUTHENTICATION.

The WITH AUTHENTICATION attribute specifies that switching the current user on a connection based on this trusted-context object to this user requires authentication.

The WITHOUT AUTHENTICATION attribute specifies that switching the current user does not require authentication.

DROP USE FOR clause

The DROP USE FOR clause specifies who can no longer use the trusted-context object. The users who are removed from the definition of the trusted-context object are those users who are currently allowed to use the trusted-context object. If one or more, but not all, users can be removed from the definition of the trusted-context object, the specified users are removed and a warning is returned. If none of the specified users can be removed from the definition of the trusted-context object, an error is returned.

If you use the DROP USE FOR clause on PUBLIC, it removes the ability of all users (except the SYSTEM AUTHID user ID, and any other individual users whose identifiers have been explicitly enabled) to use this trusted-context object.

Examples of modifying a trusted-context

For the following example, assume that trusted-context object `appserver` exists and that it is enabled. The following ALTER TRUSTED CONTEXT statement resets the object mode of the appserver trusted-context object to `DISABLE`. While in that mode, the appserver trusted context continues to exist, but it cannot be used to access the database server.

```
ALTER TRUSTED CONTEXT appserver
  DISABLE;
```

For the following example, assume that trusted-context object `secure_role` exists. Issue an ALTER TRUSTED CONTEXT statement to modify the existing user `joe` to use the trusted-context object with authentication and to add everyone else to use the trusted-context object without authentication.

```
ALTER TRUSTED CONTEXT securerole
  REPLACE USE FOR joe WITH AUTHENTICATION
  ADD USE FOR PUBLIC WITHOUT AUTHENTICATION;
```

The following example modifies the trusted-context object `securerole` to use an IPv4 address that is different from what it was originally defined to use.

```
ALTER TRUSTED CONTEXT securerole
  ALTER ATTRIBUTES (ADDRESS '9.12.155.200');
```

ALTER USER statement (UNIX™, Linux™)

Use the ALTER USER statement to change one or more of the properties, including the password, user ID, surrogate group, administrative authorization, and home directory, and to enable or disable the account of an internally authenticated user, or of the default internally authenticated user.

This statement is an extension to the ANSI/ISO standard for the SQL language.

Syntax

(explicit id unique_209_Connect_42_altus02) unique_209_Connect_42_altus02 **ALTER** { **DEFAULT USER** | **USER** *user* [{ **ACCOUNT LOCK** | **ACCOUNT UNLOCK** }] } { | { **ADD** | **MODIFY** } { **PASSWORD***password* | [**UID***user_ID*] | [**GROUP** ({ [] | *surrog_group_ID* | *surrog_group* })] | **USER***surrog_user* | [**AUTHORIZATION** ({ | **DBSA** | **DBSSO** | **AAO** | **BARGROUP** })] | [**HOME** "*directory*"] } | **DROP** { **PASSWORD** | [**UID**] | [**GROUP** ({ | *surrog_group_ID* | *surrog_group* })] | **USER** | [**AUTHORIZATION** ({ | **DBSA** | **DBSSO** | **AAO** | **BARGROUP** })] | [**HOME**] } }

Element	Description	Restrictions	Syntax
directory	Path name of directory where user files are stored.	Must be 255 bytes or fewer, and must conform to the rules of your operating system. The *directory* must also: • Belong to the mapped *user_ID* and *surrog_group_ID*. • Have read, write, and execute permissions for the owner.	Quoted String on page 1202
password	Password for internal authentication of the user.	Must be between 6 and 32 bytes.	Quoted String on page 1202

Element	Description	Restrictions	Syntax
surrog_group	Name of an existing operating system group (surrogate group) that has the permissions to which you want to map *user*. The list of *surrog_group* values must be enclosed in parentheses.	Must be 32 bytes or fewer.	Owner name on page 1262
surrog_group_ID	Group identifier number (surrogate group) to which you want to map the *user*. The list of *surrog_group_id* value or values that you specify must be enclosed in parentheses.	The *surrog_group_ID* cannot be: • A group ID with server administrative privileges (DBSA, DBSSO, AAO, and BARGROUP) • Group 0 (**root**, sometimes referred to as **wheel** or **system**) • Group 80 on Mac OS X (**admin**) • A group ID associated with group **bin** or group **sys**	Literal Number on page 1198
surrog_user	Name of an existing OS user account (surrogate user) on the Informix® host computer having the permissions to which you want to map *user*.	Must conform to the rules of your operating system	Owner name on page 1262
user	Authorization identifier of the specific user that you are mapping to user properties.	Must be an authenticated authorization identifier	Owner name on page 1262
user_ID	User identifier number to which you want to map *user*.	*user_ID* cannot be the one that belongs to user **root** or user **informix**.	Literal Number on page 1198

Usage

Only a DBSA can run the ALTER USER statement. With a non-root installation, the user who installs the server is the equivalent of the DBSA, unless the user delegates DBSA privileges to a different user.

The USERMAPPING configuration parameter must be set to a value (ADMIN or BASIC) that enables support for mapped users before users created with the CREATE USER statement can connect to the database server.

The USERMAPPING configuration parameter must be set to ADMIN to enable the AUTHORIZATION clause. For more information about this deprecated syntax, see the CREATE USER statement (UNIX, Linux) on page 416 description of the AUTHORIZATION clause.

You must also enter values in the SYSUSERMAP table of the **sysusers** database to map users with the appropriate user properties so that the mapped user statements of SQL to work correctly.

Mapped users can connect to Informix® with the surrogate user properties if they authenticate with pluggable authentication module (PAM) or single sign-on (SSO).

The best practice is to map *user* to a specific *surrog_user* that is reserved as a surrogate user identity only. You can add groups associated with the surrogate user identity with the GROUP keyword, and change the home directory with the HOME keyword.

The ALTER USER statement does not affect any active operations with the same surrogate user or user ID. Only subsequent operations that require authentication are affected.

An ALTER USER statement can add a password for a user with the ADD keyword only if that user does not have a password. To change an existing password, use the MODIFY option in the ALTER USER statement.

The total number of groups after the ALTER USER operation cannot exceed 16, which is the maximum number of allowed groups.

An ALTER USER statement can only add a home directory with the ADD keyword if no home directory exists. To modify an existing home directory, use the MODIFY keyword.

In a single ALTER USER statement, a specific property can only be specified once. For example, you cannot drop a GROUP property and add a GROUP property in the same statement.

After the ALTER USER statement, the user must have either one USER property or one UID property.

Execution of the ALTER USER statement can be audited with the ALUR audit code.

Example

Examples

Example 1: Replace a USER property with a UID property

The following statement replaces the USER property with a UID property for the user **bill**:

```
ALTER USER bill DROP USER, ADD UID 1360;
```

Example 2: Change and add properties

The following statement changes a UID property, adds the DBSA group, and adds a home directory for the user **bill**:

```
ALTER USER bill MODIFY UID 1361, ADD GROUP (dbsa), ADD HOME "/u/user1";
```

Example 3: Unlock an account and drop an authorization property

The following statement unlocks the account and drops the DBSSO authorization for the user **bill**:

```
ALTER USER bill ACCOUNT UNLOCK DROP AUTHORIZATION (dbsso);
```

Example 4: Drop a home directory

The following statement drops the home directory for the user **bill**:

```
ALTER USER bill DROP HOME;
```

BEGIN WORK statement

Use the BEGIN WORK statement to start a *transaction*, which is a series of database operations that the COMMIT WORK or ROLLBACK WORK statement terminates, and that the database server treats as a single unit of work. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_211_Connect_42_bwork001) unique_211_Connect_42_bwork001 **BEGIN WORK** [[] **WITHOUT REPLICATION**]

Usage

The BEGIN WORK statement is valid only in a database that supports transaction logging. This statement is not valid in an ANSI-compliant database.

Each row that an UPDATE, DELETE, INSERT, or MERGE statement affects during a transaction is locked and remains locked throughout the transaction. A transaction that contains many such statements or that contains statements that affect many rows can exceed the limits that your operating system or the database server configuration imposes on the number of simultaneous locks.

If no other user is accessing the table, you can avoid locking limits and reduce locking overhead by locking the table with the LOCK TABLE statement after you begin the transaction. Like other locks, this table lock is released when the transaction terminates. The example of a transaction on Example of BEGIN WORK on page 169 includes a LOCK TABLE statement.

Important: Issue the BEGIN WORK statement only if a transaction is not in progress. If you issue a BEGIN WORK statement while you are in a transaction, the database server returns an error.

The WORK keyword is optional. The following two statements are equivalent:

```
BEGIN;
BEGIN WORK;
```

In reading SQL source code that omits the WORK keyword, do not confuse the BEGIN statement of SQL with the SPL keyword BEGIN, which, together with the END keyword, can be used as a delimiter to define a statement block within an SPL routine

In , if you use the BEGIN WORK statement within a UDR called by a WHENEVER statement, specify WHENEVER SQLERROR CONTINUE and WHENEVER SQLWARNING CONTINUE before the ROLLBACK WORK statement. These statements prevent the program from looping endlessly if the ROLLBACK WORK statement encounters an error or a warning.

BEGIN WORK and ANSI-Compliant Databases

In an ANSI-compliant database, you do not need the BEGIN WORK statement because transactions are implicit; every SQL statement occurs within a transaction. The database server generates a warning when you use a BEGIN WORK statement immediately after any of the following statements:

- DATABASE
- COMMIT WORK
- CREATE DATABASE
- ROLLBACK WORK

The database server returns an error when you use a BEGIN WORK statement after any other statement in an ANSI-compliant database.

BEGIN WORK WITHOUT REPLICATION (ESQL/C)

When you use Enterprise Replication for data replication, you can use the BEGIN WORK WITHOUT REPLICATION statement to start a transaction that does not replicate to other database servers.

You cannot execute BEGIN WORK WITHOUT REPLICATION as a stand-alone embedded statement in applications. Instead you must execute this statement indirectly. You can use either of the following methods:

- You can use a combination of the PREPARE and EXECUTE statements to prepare and execute the BEGIN WORK WITHOUT REPLICATION statement.
- You can use the EXECUTE IMMEDIATE statement to prepare and execute BEGIN WORK WITHOUT REPLICATION in a single step.

You cannot use the DECLARE *cursor* CURSOR WITH HOLD statement with the BEGIN WORK WITHOUT REPLICATION statement.

For more information about data replication, see the .

Example of BEGIN WORK

When consecutive SQL statements perform what is logically a single unit of work, you can define a transaction by grouping them between the BEGIN WORK and COMMIT WORK statements. If the business requirements dictate that either all of the statements be performed successfully, or else that none of them be performed, you can enclose the statements of the transaction between BEGIN WORK to start a transaction and COMMIT WORK to complete the transaction successfully (or ROLLBACK WORK, to cancel the transaction, if the program detects an error).

In the following program fragment, the transaction locks the **stock** table (LOCK TABLE), updates rows in the **stock** table (UPDATE), deletes rows from the **stock** table (DELETE), and inserts a row into the **manufact** table (INSERT). In this example (with no error handling), the database server executes each of these SQL statements in sequence:

```
BEGIN WORK;
   LOCK TABLE stock;
   UPDATE stock SET unit_price = unit_price * 1.10
      WHERE manu_code = 'KAR';
   DELETE FROM stock WHERE description = 'baseball bat';
   INSERT INTO manufact (manu_code, manu_name, lead_time)
      VALUES ('LYM', 'LYMAN', 14);
COMMIT WORK;
```

Each statement itself is atomic; it either completes successfully or else the database is unchanged afterwards. If any of these statements fail, the other statements will still be executed and the net result is as if the failed statement was never attempted. When the COMMIT WORK statement is executed, the successful changes are made permanent.

Typically, however, transactions are defined with error handling, so that the database server must perform a sequence of operations either completely or not at all. In this case, when you include all of the operations within a single transaction, the database server guarantees that all the statements are completely and perfectly committed to disk, or else the database can be restored to the same state that it was in before the transaction began.

By adding appropriate error handling (for example, by setting the **DBACCNOIGN** environment variable in DB-Access, or by adding EXEC SQL WHENEVER ERROR STOP in ESQL/C), the transaction can be implicitly rolled back because the program stops on an error without executing COMMIT WORK. More careful conditional coding in a programming language such as ESQL/C allows the programmer to explicitly roll back the transaction while continuing the larger program.

Error-handling and business logic in applications and UDRs can also delimit one or more portions of a transaction by including SAVEPOINT and ROLLBACK TO SAVEPOINT statements. If the ROLLBACK TO SAVEPOINT statement is issued after an error is encountered, or after the results of part of the transaction indicate a conflict with a business rule or with some other criterion, only the changes that were made to the database between the ROLLBACK statement and its specified or default savepoint are cancelled, rather than the entire transaction. The current transaction continues at the statement that follows the ROLLBACK statement, with any uncommitted changes to the data or to the schema of the database from operations that preceded the savepoint remain pending, until the entire transaction is either committed or rolled back. Any locks held by statements that were rolled back are retained until the complete transaction ends.

CLOSE statement

Use the CLOSE statement when you no longer need to refer to the set of rows associated with a Select cursor or with a Function cursor. With ESQL/C, this statement can also flush and close an Insert cursor. Use this statement with or SPL.

Syntax

(explicit id unique_215_Connect_42_close001) unique_215_Connect_42_close001(explicit id unique_215_Connect_42_close002)

unique_215_Connect_42_close002 **CLOSE** { *cursor_id* | *cursor_id_var* }

Element	Description	Restrictions	Syntax
cursor_id	Name of cursor to be closed	Must have been declared	Identifier on page 1231
cursor_id_var	Host variable that contains the value of *cursor_id*	Must be of a character data type	Must conform to language-specific rules for names.

Usage

Closing a cursor makes the cursor unusable in any statements except OPEN or FREE and releases resources that the database server had allocated to the cursor.

In a database that is not ANSI-compliant, you can close a cursor that has not been opened or that has already been closed. No action is taken in these cases.

In an ANSI-compliant database, the database server returns an error if you close a cursor that was not open.

Examples

The following statement closes the cursor, *democursor*.

```
EXEC SQL close democursor;
```

The following is ESQL/C Source code example from demo1.ec:

```
#include <stdio.h>

EXEC SQL define FNAME_LEN       15;
EXEC SQL define LNAME_LEN       15;

main()
{

EXEC SQL BEGIN DECLARE SECTION;
    char fname[ FNAME_LEN + 1 ];
    char lname[ LNAME_LEN + 1 ];
EXEC SQL END DECLARE SECTION;

    printf( "DEMO1 Sample ESQL Program running.\n\n");

    EXEC SQL WHENEVER ERROR STOP;

    EXEC SQL connect to 'stores7';

    EXEC SQL declare democursor cursor for
        select fname, lname
           into :fname, :lname
           from customer
           where lname < "C";

    EXEC SQL open democursor;
    for (;;)
        {
```

```
        EXEC SQL fetch democursor;
        if (strncmp(SQLSTATE, "00", 2) != 0)
            break;
        printf("%s %s\n",fname, lname);
        }
    if (strncmp(SQLSTATE, "02", 2) != 0)
        printf("SQLSTATE after fetch is %s\n", SQLSTATE);

    EXEC SQL close democursor;
    EXEC SQL free democursor;
    EXEC SQL create routine from 'del_ord.sql';
    EXEC SQL disconnect current;
    printf("\nDEMO1 Sample Program over.\n\n");
   exit(0);
}
```

Closing a Select or Function Cursor

When a cursor is associated with a SELECT, EXECUTE FUNCTION, or EXECUTE PROCEDURE statement of SQL, closing the cursor terminates the associated SQL statement.

The database server releases all resources that it might have allocated to the active set of rows, for example, a temporary table that the cursor used to hold an ordered set. The database server also releases any locks that it might have held on rows that were selected through the cursor. If a transaction contains the CLOSE statement, however, the database server does not release the locks until you issue the COMMIT WORK or ROLLBACK WORK statement.

After you close a Select cursor or a Function cursor, the FETCH statement cannot reference that cursor until you reopen it.

In an SPL routine, the built-in **SQLCODE** function can indicate the result of the CLOSE statement for a Select cursor or a Function cursor. This function returns a value equivalent to the **SQLCODE** field of the **sqlca** structure. Informix® issues an error, however, if you invoke the built-in **SQLCODE** function outside the calling context of an SPL routine.

Closing an Insert Cursor

Because Informix® does not support Insert cursors in SPL routines, the discussion of Insert cursors in this section applies only to . In SPL routines, the CLOSE statement can reference only a Select cursor or a Function cursor that the DECLARE statement defined. (A FOREACH statement of SPL that has an INSERT statement in its statement block can declare a *direct cursor* that functionally resembles an Insert cursor, but the CLOSE statement cannot reference a direct cursor that FOREACH declared. Informix® closes the direct cursor automatically at runtime when program control exits from the FOREACH loop where the direct cursor was defined.)

In , the CLOSE statement treats a cursor that is associated with an INSERT statement differently from one that is associated with a SELECT, EXECUTE FUNCTION, or EXECUTE PROCEDURE statement. When a cursor identifier is associated with an INSERT statement, the CLOSE statement writes any remaining buffered rows into the database. The number of rows that were successfully inserted into the database is returned in the third element of the **sqlerrd** array, **sqlca.sqlerrd[2]**, in the **sqlca** structure. For information on how to use **SQLERRD** to count the total number of rows that were inserted, see Error Checking on page 640.

The **SQLCODE** field of the **sqlca** structure indicates the result of the CLOSE statement for an Insert cursor. If all buffered rows are successfully inserted, **SQLCODE** is set to zero. If an error is encountered, the **SQLCODE** field is set to a negative error message number.

When **SQLCODE** is zero, the row buffer space is released, and the cursor is closed; that is, you cannot execute a PUT or FLUSH statement that names the cursor until you reopen it.



Tip: When you encounter an **sqlca.SQLCODE** error, a corresponding **SQLSTATE** error value also exists. For information about how to get the message text, check the GET DIAGNOSTICS statement.

If the insert is not successful, the number of successfully inserted rows is stored in **sqlerrd**. Any buffered rows that follow the last successfully inserted row are discarded. Because the insert fails, the CLOSE statement fails also, and the cursor is not closed. For example, a CLOSE statement can fail if insufficient disk space prevents some of the rows from being inserted. In this case, a second CLOSE statement can be successful because no buffered rows exist. An OPEN statement can also be successful because the OPEN statement performs an implicit close.

Closing a Collection Cursor

You can declare both Select and Insert cursors on collection variables. Such cursors are called Collection cursors. Use the CLOSE statement to deallocate resources that have been allocated for the Collection cursor. Only ESQL/C routines can use CLOSE to reference Insert cursors on collection variables. The CLOSE statement in SPL routines cannot reference direct Collection cursors that the FOREACH statement of SPL can declare.

For more information on how to use a Collection cursor, see Fetching from a Collection Cursor on page 520 and Inserting into a Collection Cursor on page 639.

Using End of Transaction to Close a Cursor

The COMMIT WORK and ROLLBACK WORK statements close all cursors except those that are declared with a hold. It is better to close all cursors explicitly, however. For Select or Function cursors, this action simply makes the intent of the program clear. It also helps to avoid a logic error if the WITH HOLD clause is later added to the declaration of a cursor.

For an Insert cursor in ESQL/C routines, it is important to use the CLOSE statement explicitly so that you can test the error code. Following the COMMIT WORK statement, **SQLCODE** reflects the result of the COMMIT statement, not the result of closing cursors. If you use a COMMIT WORK statement without first using a CLOSE statement, and if an error occurs while the last buffered rows are being written to the database, the transaction is still committed.

For how to use Insert cursors and the WITH HOLD clause, see DECLARE statement on page 433.

In an ANSI-compliant database, a cursor cannot be closed implicitly. You must issue the CLOSE statement.

CLOSE DATABASE statement

Use the CLOSE DATABASE statement to close the implicit connection to the current database. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
CLOSE DATABASE
```

Usage

When you issue a CLOSE DATABASE statement, you can issue only the following SQL statements immediately after it:

- CONNECT
- CREATE DATABASE
- DATABASE
- DROP DATABASE
- DISCONNECT

  (The DISCONNECT statement is valid here only if an explicit connection existed before CLOSE DATABASE was executed.)

Issue the CLOSE DATABASE statement before you drop the current database.

If your current database supports transaction logging, and if you have started a transaction, you must issue a COMMIT WORK or ROLLBACK WORK statement before you can use the CLOSE DATABASE statement.

The following example shows how to use the CLOSE DATABASE statement before you drop the current database to which your session had established an implicit connection:

```
DATABASE stores_demo;
. . .
CLOSE DATABASE;
DROP DATABASE stores_demo;
```

In , the CLOSE DATABASE statement cannot appear in a multistatement PREPARE operation.

If a previous CONNECT statement has established an explicit connection to a database, and that connection is still your current connection, you cannot use the CLOSE DATABASE statement to close that explicit connection. (You can use the DISCONNECT statement to close the explicit connection.)

If you use the CLOSE DATABASE statement within a UDR called by a WHENEVER statement, specify WHENEVER SQLERROR CONTINUE and WHENEVER SQLWARNING CONTINUE before the ROLLBACK WORK statement. This action prevents the program from looping endlessly if the ROLLBACK WORK statement encounters an error or a warning.

When you issue the CLOSE DATABASE statement, any declared cursors are no longer valid. You must re-declare any cursors that you want to use.

In an ANSI-compliant database, if no error is encountered while you exit from in non-interactive mode without issuing the CLOSE DATABASE, COMMIT WORK, or DISCONNECT statement, the database server automatically commits any open transaction.

COMMIT WORK statement

Use the COMMIT WORK statement to commit all modifications made to the database from the beginning of a transaction.

Syntax

```
COMMIT [ WORK ]
```

Usage

The COMMIT WORK statement informs the database server that you reached the end of a series of statements that must succeed as a single unit. The database server takes the required steps to make sure that all modifications that the transaction makes are completed correctly and saved to disk.

Use COMMIT WORK only at the end of a multistatement operation in a database with transaction logging, when you are sure that you want to keep all changes made to the database from the beginning of a transaction.

The COMMIT WORK statement releases all row and table locks.

The WORK keyword is optional in a COMMIT WORK statement. The following two statements are equivalent:

```
COMMIT;
COMMIT WORK;
```

The following example shows a transaction bounded by BEGIN WORK and COMMIT WORK statements.

```
BEGIN WORK;
   DELETE FROM call_type WHERE call_code = 'O';
   INSERT INTO call_type VALUES ('S', 'order status');
COMMIT WORK;
```

In this example, the user first deletes the row from the **call_type** table where the value of the **call_code** column is `O`. The user then inserts a new row in the **call_type** table where the value of the **call_code** column is `S`. The database server guarantees that both operations succeed or else neither succeeds.

In , the COMMIT WORK statement closes all open cursors except those that were declared using the WITH HOLD option.

Issuing COMMIT WORK in a Database That Is Not ANSI Compliant

In a database that is not ANSI compliant, but that supports transaction logging, if you initiate a transaction with a BEGIN WORK statement, you must issue a COMMIT WORK statement at the end of the transaction. If you fail to issue a COMMIT WORK statement in this case, the database server rolls back any modifications that the transaction made to the database.

If you do not issue a BEGIN WORK statement, however, each statement executes within its own transaction. These single-statement transactions do not require either a BEGIN WORK statement or a COMMIT WORK statement.

Explicit DB-Access Transactions

When you use DB-Access in interactive mode with a database that is not ANSI-compliant but that supports transaction logging, if you select the **Commit** menu but do not issue the COMMIT WORK statement after a transaction has been started by the BEGIN WORK statement, DB-Access automatically commits the data, but issues the following warning:

```
Warning: Data commit is a result of unhandled exception in TXN PROC/FUNC
```

The purpose of this warning is to remind you to issue COMMIT WORK explicitly to end a transaction that BEGIN WORK initiated.

In non-interactive mode, however, DB-Access rolls back the current transaction if you end a session without issuing the COMMIT WORK statement.

Issuing COMMIT WORK in an ANSI-Compliant Database

In an ANSI-compliant database, you do not need BEGIN WORK to mark the beginning of a transaction. You only need to mark the end of each transaction, because a transaction is always in effect. A new transaction starts automatically after each COMMIT WORK or ROLLBACK WORK statement.

You must, however, issue an explicit COMMIT WORK statement to mark the end of each transaction. If you fail to do so, the database server rolls back any modifications that the transaction made to the database.

In an ANSI-compliant database, however, if no error is encountered while you exit from in non-interactive mode without issuing the CLOSE DATABASE, COMMIT WORK, or DISCONNECT statement, the database server automatically commits any open transaction.

CONNECT statement

Use the CONNECT statement to connect to a database environment. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_229_Connect_42_connect001) unique_229_Connect_42_connect001(explicit id unique_229_Connect_42_connect002) unique_229_Connect_42_connect002(explicit id unique_229_Connect_42_connect003) unique_229_Connect_42_connect003(explicit id unique_229_Connect_42_connect004) unique_229_Connect_42_connect004 **CONNECT TO** { <Database Environment> [[<USER Clause> [**TRUSTED**]] | **DEFAULT** } [**WITH CONCURRENT TRANSACTION**]

Usage

The CONNECT statement connects an application to a *database environment*, which can be a database, a database server, or a database and a database server. If the application successfully connects to the specified database environment, the connection becomes the *current connection* for the application. The SQL statements fail if the application has no current connection to a database server. If you specify a database name, the database server opens that database. You cannot include CONNECT within a PREPARE statement.

An application can connect to several database environments at the same time, and it can establish multiple connections to the same database environment, provided each connection has a unique connection name.

On UNIX™, the only restriction on establishing multiple connections to the same database environment is that an application can establish only one connection to each local server that uses the shared-memory connection mechanism. To find out whether a local server uses the shared-memory connection mechanism or the local-loopback connection mechanism,

examine the `$INFORMIXDIR/etc/sqlhosts` file. For more information on the `sqlhosts` file, refer to your *HCL® Informix® Administrator's Guide*.

On Windows™, the local connection mechanism is named pipes. Multiple connections to the local server from one client can exist.

Only one connection is current at any time; other connections are dormant. The application cannot interact with a database through a dormant connection. When an application establishes a new connection, that connection becomes current, and the previous current connection becomes dormant. You can make a dormant connection current with the SET CONNECTION statement. See also SET CONNECTION statement on page 779.

For connections between databases of different Informix® instances, you cannot establish multiple active connections between the same two database servers using different server aliases. At any time, there can be only one active connection from the local server to a remote server. If you use CONNECT TO *dbserveralias* statements to specify different server aliases to connect to the same remote server, where the *dbserveralias* identifiers are declared in setting of the DBSERVERALIASES configuration parameter setting, no error message is issued, but the initial connection is reused.

Privileges for Executing the CONNECT Statement

The current user, or PUBLIC, must hold the Connect privilege on the database that the CONNECT statement specifies. The user who executes the CONNECT statement cannot have the same authorization identifier as an existing role in that database.

For information on how to use the USER Authentication clause to specify an alternate user name when the CONNECT statement connects to a database server on a remote host, see USER Authentication Clause on page 180.

Connection Context

Each connection encompasses a set of information that is called the *connection context*. The connection context includes the name of the current user, the information that the database environment associates with this name, and information on the state of the connection (such as whether an active transaction is associated with the connection). The connection context is saved when an application becomes dormant, and this context is restored when the application becomes current again. (For more information, see Making a dormant connection as the current connection on page 779.)

Database Environment

The CONNECT statement, like the SET CONNECTION statement, can use the Database Environment syntax segment to specify the database or the database server to which the application is attempting to establish a connection. Unlike the SET CONNECTION statement, the CONNECT statement can also declare a name for this connection to the specified database environment.

(explicit id unique_230_Connect_42_dbenv001) unique_230_Connect_42_dbenv001

Database Environment

" { *'dbname'* | *'@dbservername'* | *'dbname@dbservername'* | *db_var* } [AS { *'connection'* | *connection_var* }] "

Element	Description	Restrictions	Syntax
connection	Optional case-sensitive name that you declare here for a connection	Must be unique among connection names	Identifier on page 1231
connection_var	Host variable that stores the name of *connection*	Must be a fixed-length character data type	Language specific
db_var	Host variable that contains a valid database environment (in one of the formats in the syntax diagram)	Must be a fixed-length character data type, whose contents are in a format from the syntax diagram	Language specific
dbname	Database to which to connect	Must already exist	Identifier on page 1231
dbservername	Name of the database server to which a connection is made	Must already exist; blank space is not valid between @ symbol and *dbservername*. See also Restrictions on dbservername on page 178.	Identifier on page 1231

If the **DELIMIDENT** environment variable is set, any quotation (') marks in the database environment must be single. If **DELIMIDENT** is not set, then either single (') or double (") quotation marks are valid here.

Restrictions on dbservername

If you specify *dbservername*, it must satisfy the following restrictions.

- If the database server that you specify is not online, you receive an error.
- The database server that you specify in *dbservername* must match the name of a database server in the **sqlhosts** file.

Note: If the name of a database server is a delimited identifier, or if it includes uppercase letters, that database server cannot participate in cross-server distributed DML operations. (If the server name includes uppercase letters, it also cannot participate in cross-database distributed DML operations by SQL statements that specify the server name as a qualifier to a database name. These statements fail with error -908, because the SQL parser downshifts all uppercase letters in server names to lowercase characters.) To avoid this restriction, specify only undelimited

names with no uppercase letters when you declare the name or the alias of a database server that will participate in distributed queries.

Specifying the Database Environment

You can specify a database server and a database, or a database server only, or a database only. How a database is located and opened depends on whether you specify a database server name in the database environment expression.

Only Database Server Specified

The *@dbservername* option establishes a connection to the database server only; it does not open a database. When you use this option, you must subsequently use the DATABASE or CREATE DATABASE statement (or a PREPARE statement for one of these statements and an EXECUTE statement) to open a database.

Database Server and Database Specified

If you specify both a database server and a database, your application connects to the database server, which locates and opens the database.

Only Database Specified

The *dbname* option establishes a connection to the default database server or to another database server in the **DBPATH** environment variable. It also locates and opens the specified database. (The same is true of the *db_var* option if this specifies only a database name.)

If you specify only *dbname*, its database server is read from the **DBPATH** environment variable. The database server in the **INFORMIXSERVER** environment variable is always added before the **DBPATH** value.

On UNIX™, set the **INFORMIXSERVER** and **DBPATH** environment variables as the following example (for the C shell) shows:

```
setenv INFORMIXSERVER srvA
setenv DBPATH //srvB://srvC
```

On Windows™, choose **Start > Programs > Informix® > setnet32** from the Task Bar and set the **INFORMIXSERVER** and **DBPATH** environment variables:

```
set INFORMIXSERVER = srvA
set DBPATH = //srvA://srvB://srvC
```

The next example shows the resulting **DBPATH** that your application uses:

```
//srvA://srvB://srvC
```

The application first establishes a connection to the database server that **INFORMIXSERVER** specifies. The database server uses parameters in the configuration file to locate the database. If the database does not reside on the default database server, or if the default database server is not online, the application connects to the next database server in **DBPATH**. In the previous example, that database server would be **srvB**.

Declaring a Connection Name

In ESQL/C applications, you can declare an identifier for the connection to the database environment by including the AS keyword, followed by a quoted string or by a host variable that stores the identifier. The host variable must be a fixed-length character data type.

Connection Identifiers

The optional *connection* name is a unique identifier that an ESQL/C application can use to refer to a connection in subsequent SET CONNECTION and DISCONNECT statements. If the application does not provide a connection name (or a connection host variable), it can refer to the connection using the database environment. If the application makes more than one connection to the same database environment, however, each connection must have a unique name.

Only the CONNECT statement can use the AS keyword to declare a connection name. The CONNECT statement cannot, however, reference a previously declared connection name to specify a connection to a database environment.

USER Authentication Clause

The USER Authentication clause specifies information that is used to determine whether the application can access the target computer on a remote host.

USER Authentication Clause

" **USER** { *'user_id'* | *user_id_var* } "

" **USING***validation_var* "

Element	Description	Restrictions	Syntax
user_id	Valid login name	See Restrictions on the User Identifier Parameter on page 181.	Quoted String on page 1202
user_id_var	Host variable that contains *user_id*	Must be a fixed-length character data type; same restrictions as *user_id*	Language specific
validation_var	Host variable that contains a valid password for login name in *user_id* or *user_id_var*	Must be a fixed-length character type. See Restrictions on the Validation Variable Parameter on page 181.	Language specific

The USER Authentication clause is required when the CONNECT statement connects to the database server on a remote host. Subsequent to the CONNECT statement, all database operations on the remote host use the specified user name.

In DB-Access, the USING clause is valid within files executed from DB-Access. In interactive mode, DB-Access prompts you for a password, so the USING keyword and *validation_var* are not used.

Restrictions on the Validation Variable Parameter

On UNIX™, the password stored in *validation_var* must be a valid password and must exist in the **/etc/passwd** file. If the application connects to a remote database server, the password must exist in this file on both the local and remote database servers.

On Windows™, the password stored in *validation_var* must be valid and must be the password entered in **User Manager**. If the application connects to a remote database server, the password must exist in the domain of both the client and the server.

Restrictions on the User Identifier Parameter

The connection is rejected if any of the following conditions occur:

- The specified user lacks the privileges to access the database specified in the database environment.
- The specified user lacks the permissions to connect to the remote host.
- You supply a USER Authentication clause but omit the USING *validation_var* specification.

In compliance with the X/Open standard for the CONNECT statement, the preprocessor supports a CONNECT statement that has a USER Authentication clause without the USING *validation_var* specification. If the *validation_var* is not present, however, the database server rejects the connection at runtime.

On UNIX™, the *user_id* that you specify must be a valid login name and must exist in the **/etc/passwd** file. If the application connects to a remote server, the login name must exist in this file on both the local and remote database servers.

On Windows™, the *user_id* that you specify must be a valid login name and must exist in **User Manager**. If the application connects to a remote server, the login name must exist in the domain of both the client and the server.

Use of the Default User ID

If you do not supply the USER Authentication clause, the default user ID is used to attempt the connection.

The default user ID is the login name of the user running the application. In this case, you obtain network permissions with the standard authorization procedures. For example, on UNIX™, the default user ID must match a user ID in the trusted-host file (the `/etc/hosts.equiv` file or the file specified by the REMOTE_SERVER_CFG configuration parameter). On Windows™, you must be a member of the domain, or if the database server is installed locally, you must be a valid user on the computer where it is installed.

The DEFAULT Connection Specification

Instead of specifying an explicit database environment, you can use the DEFAULT keyword to request a *default connection* to a default database server. The default database server can be local or remote. To designate the default database server, set its name in the **INFORMIXSERVER** environment variable. This option of CONNECT does not open a database.

If the CONNECT TO DEFAULT statement succeeds, you must use the DATABASE statement or the CREATE DATABASE statement to open or create a database in the default database environment.

The Implicit Connection with DATABASE Statements

If you do not execute a CONNECT statement in your application, the first SQL statement must be one of the following *database statements* (or a single statement PREPARE for one of the following statements):

- DATABASE
- CREATE DATABASE
- DROP DATABASE

If one of these database statements is the first SQL statement in an application, the statement establishes a connection to a database server, which is known as an *implicit* connection. If the database statement specifies only a database name, the database server name is obtained from the **DBPATH** environment variable. This situation is described in Specifying the Database Environment on page 179.

An application that makes an implicit connection can establish other connections explicitly (using the CONNECT statement) but cannot establish another implicit connection unless the original implicit connection is closed. An application can terminate an implicit connection using the DISCONNECT statement. After you create an explicit connection, you cannot use any database statement to create implicit connections until after you close the explicit connection.

After *any* implicit connection is made, that connection is considered to be the default connection, regardless of whether the database server is the default that the **INFORMIXSERVER** environment variable specifies. This feature allows the application to refer to the implicit connection if additional explicit connections are made, because the implicit connection has no identifier.

For example, if you establish an implicit connection followed by an explicit connection, you can make the implicit connection current by issuing the SET CONNECTION DEFAULT statement. This means, however, that once you establish an implicit connection, you cannot use the CONNECT DEFAULT statement, because the implicit connection is now the default connection.

The database statements can always be used to open a database or create a new database on the current database server.

WITH CONCURRENT TRANSACTION Option

The WITH CONCURRENT TRANSACTION clause enables you to switch to a different connection while a transaction is active in the current connection. If the current connection was *not* established using the WITH CONCURRENT TRANSACTION clause, you cannot switch to a different connection if a transaction is active; the CONNECT or SET CONNECTION statement fails, returning an error, and the transaction in the current connection continues to be active.

In this case, the application must commit or roll back the active transaction in the current connection before it switches to a different connection.

The WITH CONCURRENT TRANSACTION clause supports the concept of multiple concurrent transactions, where each connection can have its own transaction and the COMMIT WORK and ROLLBACK WORK statements affect only the current connection. The WITH CONCURRENT TRANSACTION clause does not support global transactions in which a single

transaction spans databases over multiple connections. The COMMIT WORK and ROLLBACK WORK statements do not act on databases across multiple connections.

The following example illustrates how to use the WITH CONCURRENT TRANSACTION clause:

```
main()
{
EXEC SQL connect to 'a@srv1' as 'A';
EXEC SQL connect to 'b@srv2' as 'B' with concurrent transaction;
EXEC SQL connect to 'c@srv3' as 'C' with concurrent transaction;

/*
   Execute SQL statements in connection 'C' , starting a transaction
*/
EXEC SQL set connection 'B'; -- switch to connection 'B'

/*
   Execute SQL statements starting a transaction in 'B'.
   Now there are two active transactions, one each in 'B' and 'C'.
*/

EXEC SQL set connection 'A'; -- switch to connection 'A'

/*
   Execute SQL statements starting a transaction in 'A'.
   Now there are three active transactions, one each in 'A', 'B' and 'C'.
*/

EXEC SQL set connection 'C'; -- ERROR, transaction active in 'A'

/*
   SET CONNECTION 'C' fails (current connection is still 'A')
   The transaction in 'A' must be committed or rolled back because
   connection 'A' was started without the CONCURRENT TRANSACTION
   clause.
*/
```

```
EXEC SQL commit work;   -- commit tx in current connection ('A')

/*
   Now, there are two active transactions, in 'B' and in 'C',
   which must be committed or rolled back separately
*/

EXEC SQL set connection 'B'; -- switch to connection 'B'
EXEC SQL commit work;        -- commit tx in current connection ('B')

EXEC SQL set connection 'C'; -- go back to connection 'C'
EXEC SQL commit work;         -- commit tx in current connection ('C')

EXEC SQL disconnect all;
}
```

Warning: When an application uses the WITH CONCURRENT TRANSACTION clause to establish multiple connections to the same database environment, a deadlock condition can occur.

TRUSTED clause

Use the TRUSTED clause to specify that the application can connect to a database environment as a trusted connection.

CREATE ACCESS_METHOD statement

Use the CREATE ACCESS_METHOD statement to register a new primary or secondary access method in the **sysams** system catalog table.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
(explicit id unique_250_Connect_42_cream002) unique_250_Connect_42_cream002 CREATE { SECONDARY | PRIMARY } ACCESS_METHOD [IF NOT
EXISTS] access_method ( <Purpose Options>[] )
```

Element	Description	Restrictions	Syntax
access method	Name declared here for the new access method	Must be unique among access-method names in the **sysams** system catalog table	Identifier on page 1231

Usage

The CREATE ACCESS_METHOD statement adds a user-defined access method to a database. To create an access method, you specify *purpose functions* (or *purpose methods*), *purpose flags*, or *purpose values* as attributes of the access method, and you associate keywords (based on column names in the **sysams** system catalog table) with UDRs. You must have the DBA or Resource privilege to create an access method.

For information on setting purpose options, including a list of all the purpose function keywords, refer to Purpose Options on page 1266.

The PRIMARY keyword specifies a user-defined primary-access method for a virtual table. The SECONDARY keyword specifies creating a user-defined secondary-access method for a virtual index. The SECONDARY keyword (and creating virtual indexes) is not supported in the .

The following statement creates a secondary-access method named **T_tree**:

```
CREATE SECONDARY ACCESS_METHOD T_tree
(
am_getnext = ttree_getnext,

. . .
am_unique,
am_cluster,
am_sptype = 'S'
);
```

In the preceding example, the **am_getnext** keyword in the Purpose Options list is associated with the **ttree_getnext( )** UDR as the name of a method to scan for the next item that satisfies a query. This example indicates that the **T_tree** secondary access method supports unique keys and clustering, and resides in an sbspace.

Any UDR that the CREATE ACCESS_METHOD statement associates with the keyword for a purpose function task, such as the association of **ttree_getnext( )** with **am_getnext** in the preceding example, must already have been registered in the database by the CREATE FUNCTION statement (or by a functionally equivalent statement, such as CREATE PROCEDURE FROM).

The following statement creates a primary-access method named **am_tabprops** that resides in an extspace.

```
CREATE PRIMARY ACCESS_METHOD am_tabprops
(
am_open = FS_open,
am_close = FS_close,
am_beginscan = FS_beginScan,
am_create = FS_create,
am_scancost = FS_scanCost,
am_endscan = FS_endScan,
am_getnext = FS_getNext,
am_getbyid = FS_getById,
am_drop = FS_drop,
am_truncate = FS_truncate,
am_rowids,
am_sptype = 'x'
);
```

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if an access method of the specified name is already registered in the current database.

CREATE AGGREGATE statement

Use the CREATE AGGREGATE statement to create a new aggregate function and register it in the **sysaggregates** system catalog table.

User-defined aggregates (UDA) extend the functionality of the database server by performing aggregate calculations that the user implements.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_251_Connect_42_creag002) unique_251_Connect_42_creag002 **CREATE AGGREGATE [IF NOT EXISTS]** [<Owner Name> .]

aggregate **WITH** (<Modifiers>)

Modifiers

" { **INIT**=*init_func* | **ITER**=*iter_func* | **COMBINE**=*comb_func* | { **FINAL**=*final_func* | **HANDLESNULLS** } } "

Element	Description	Restrictions	Syntax
aggregate	Name of the new aggregate	Must be unique among names of built-in aggregates and UDRs	Identifier on page 1231
comb_func	Function that merges one partial result into the other and returns the updated partial result	Must specify the combined function both for parallel queries and for sequential queries	Identifier on page 1231
final_func	Function that converts a partial result into the result type	If this is omitted, then the returned value is the final result of *iter_func*	Identifier on page 1231
init_func	Function that initializes the data structures required for the aggregate computation	Must be able to handle NULL arguments	Identifier on page 1231
iter_func	Function that merges a single value with a partial result and returns updated partial result	Must specify an iterator function. If *init_func* is omitted, *iter_func* must be able to handle NULL arguments	Identifier on page 1231

Usage

You can specify the INIT, ITER, COMBINE, FINAL, and HANDLESNULLS modifiers in any order.

Important: You must specify the ITER and COMBINE modifiers in a CREATE AGGREGATE statement. You do not need to specify the INIT, FINAL, and HANDLESNULLS modifiers in a CREATE AGGREGATE statement.

The ITER, COMBINE, FINAL, and INIT modifiers specify the support functions for a user-defined aggregate. These support functions do not need to exist at the time when you create the user-defined aggregate.

If you omit the HANDLESNULLS modifier, rows with NULL aggregate argument values do not contribute to the aggregate computation. If you include the HANDLESNULLS modifier, you must define all the support functions to handle NULL values as well.

Important: A SELECT statement can include no more than one UDA expression whose first argument is the DISTINCT or UNIQUE keyword (rather than the ALL keyword, or no keyword). In a query that includes subqueries, however, you can specify either zero or one DISTINCT or UNIQUE user-defined aggregate expression at each level of the query. Built-in aggregates are not subject to this restriction.

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if an aggregate of the specified name is already registered in the current database.

Extending the Functionality of Aggregates

Informix® provides two ways to extend the functionality of aggregates. Use the CREATE AGGREGATE statement only for the second of the two cases.

- Extensions of built-in aggregates

  A built-in aggregate is an aggregate that the database server provides, such as **COUNT**, **SUM**, or **AVG**. These only support built-in data types. To extend a built-in aggregate so that it supports a user-defined data type (UDT), you must create user-defined routines that overload the binary operators for that aggregate. For further information on extending built-in aggregates, see the *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

- Creation of user-defined aggregates

  A user-defined aggregate is an aggregate that you define to perform an aggregate computation that the database server does not provide. You can use user-defined aggregates with built-in data types, extended data types, or both. To create a user-defined aggregate, use the CREATE AGGREGATE statement. In this statement, you name the new aggregate and specify the support functions that compute the aggregate result. These support functions perform initialization, sequential aggregation, combination of results, and type conversion.

Example of Creating a User-Defined Aggregate

The following example defines a user-defined aggregate named **average:**

```
CREATE AGGREGATE average
   WITH (
      INIT = average_init,
      ITER = average_iter,
      COMBINE = average_combine,
      FINAL = average_final
      );
```

Before you use the **average** aggregate in a query, you must also use CREATE FUNCTION statements to create the support functions specified in the CREATE AGGREGATE statement.

The following table gives an example of the task that each support function might perform for **average**.

Keyword	Support Function	Effect
INIT	**average_init**	Allocates and initializes an extended data type storing the current sum and the current row count
ITER	**average_iter**	For each row, adds the value of the expression to the current sum and increments the current row count by one
COMBINE	**average_combine**	Adds the current sum and the current row count of one partial result to the other and returns the updated result
FINAL	**average_final**	Returns the ratio of the current sum to the current row count and converts this ratio to the result type

Parallel Execution

The database server can break up an aggregate computation into several pieces and compute them in parallel.

The database server uses the INIT and ITER support functions to compute each piece sequentially. Then the database server uses the COMBINE function to combine the partial results from all the pieces into a single result value. Whether an aggregate is parallel is an optimization decision that is transparent to the user.

CREATE CAST statement

Use the CREATE CAST statement to register a cast that converts data from one data type to another.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
CREATE { EXPLICIT | [IMPLICIT] } CAST [IF NOT EXISTS] ( source_type AS target_type [ WITH function ] )
```

Element	Description	Restrictions	Syntax
function	UDR that you register to implement the cast	See WITH Clause on page 190.	Identifier on page 1231
source_type	Data type to be converted	Must exist in the database at the time the cast is registered. See also Source and Target Data Types on page 189.	Data Type on page 985
target_type	Data type that results from the conversion	The same restrictions that apply for the *source_type* (as listed above) also apply for the *target_type*	Data Type on page 985

Usage

A cast is a mechanism that the database server uses to convert one data type to another. The database server uses casts to perform the following tasks:

- To compare two values in the WHERE clause of a SELECT, UPDATE, or DELETE statement
- To pass values as arguments to user-defined routines
- To return values from user-defined routines

To create a cast, you must have the necessary privileges on both the *source* data type and the *target* data type. All users have access privileges to use the built-in data types. To create a cast to or from an OPAQUE, DISTINCT, or named ROW data type, however, requires the Usage privilege on that data type.

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if a cast between the two specified data types is already registered in the current database.

The CREATE CAST statement registers a cast in the **syscasts** system catalog table. For more information on **syscasts**, see the chapter on system catalog tables in the *HCL® Informix® Guide to SQL: Reference*.

Source and Target Data Types

The CREATE CAST statement defines a cast that converts a *source* type to a *target* type. Both the *source* and *target* data types must exist in the database when you execute the CREATE CAST statement to register the cast.

The *source* and the *target* data types have the following restrictions:

- Either the *source* or the *target* type, but not both, can be a built-in data type.
- Neither the *source* nor the *target* type can be a DISTINCT type of the other.
- Neither the *source* nor the *target* types can be a COLLECTION data type.

Explicit and Implicit Casts

Processing queries with multiple data types often requires casts that convert data from one data type to another.

You can use the CREATE CAST statement to create the following kinds of casts:

- Use the CREATE EXPLICIT CAST statement to define an *explicit* cast.
- Use the CREATE IMPLICIT CAST statement to define an *implicit* cast.

The database server invokes built-in casts to convert from one built-in data type to another built-in type that is not directly substitutable. For example, the database server performs conversion of a character type such as CHAR to a numeric type such as INTEGER through a built-in cast.

Explicit Casts

An explicit cast is a cast that you must specifically invoke, with either the CAST AS keywords or with the cast operator (::). The database server does *not* automatically invoke an explicit cast to resolve data type conversions. The EXPLICIT keyword is optional; by default, the CREATE CAST statement creates an explicit cast.

The following CREATE CAST statement defines an explicit cast from the **rate_of_return** opaque data type to the **percent** distinct data type:

```
CREATE EXPLICIT CAST (rate_of_return AS percent
   WITH rate_to_prcnt);
```

The following SELECT statement explicitly invokes this explicit cast in its WHERE clause to compare the **bond_rate** column (of type **rate_of_return**) to the **initial_APR** column (of type **percent**):

```
SELECT bond_rate FROM bond
   WHERE bond_rate::percent > initial_APR;
```

Implicit Casts

An implicit cast is a cast that the database server can invoke automatically when it encounters data types that cannot be compared with built-in casts. This type of cast enables the database server to automatically handle conversions between other data types.

To define an implicit cast, specify the IMPLICIT keyword in the CREATE CAST statement. For example, the following CREATE CAST statement specifies that the database server should automatically use the **prcnt_to_char( )** function to convert from the CHAR data type to a distinct data type, **percent**:

```
CREATE IMPLICIT CAST (CHAR AS percent WITH char_to_prcnt);
```

This cast only supports automatic conversion *from* the CHAR data type *to* **percent**. For the database server to convert *from* **percent** *to* CHAR, you also need to define another implicit cast, as follows:

```
CREATE IMPLICIT CAST (percent AS CHAR WITH prcnt_to_char);
```

The database server automatically invokes the **char_to_prcnt( )** function to evaluate the WHERE clause of the following SELECT statement:

```
SELECT commission FROM sales_rep WHERE commission > "25
```

Users can also invoke implicit casts explicitly. For more information on how to explicitly invoke a cast function, see Explicit Casts on page 189.

When a built-in cast does not exist for conversion between data types, you can create user-defined casts to make the necessary conversion.

WITH Clause

The WITH clause of the CREATE CAST statement specifies the name of the user-defined function to invoke to perform the cast. This function is called the cast function.

You must specify a *function name* unless the *source data type* and the *target data type* have identical representations. Two data types have identical representations when the following conditions are met:

- Both data types have the same length and alignment.
- Both data types are passed by reference or both are passed by value.

The cast function must be registered in the same database as the cast at the time the cast is invoked, but need not exist when the cast is created. The CREATE CAST statement does not check privileges on the specified *function name*, or even verify that the cast function exists. Each time a user invokes the cast explicitly or implicitly, the database server verifies that the user has the Execute privilege on the cast function.

CREATE DATABASE statement

Use the CREATE DATABASE statement to create a new database.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
CREATE DATABASE [IF NOT EXISTS] database [ INdbspace ] [ WITH { [ BUFFERED ] LOG  | LOG MODE ANSI } ] { NLSCASE SENSITIVE |
[NLSCASE INSENSITIVE] }
```

Element	Description	Restrictions	Syntax
database	Name that you declare here for the new database that you are creating	Must be unique among names of databases of the database server	Database Name on page 1224
dbspace	The dbspace to store the data for this database; default is the **root** dbspace	Must exist Cannot be the name of a dbspace that is dedicated to a tenant database.	Identifier on page 1231

Usage

This statement is an extension to ANSI-standard syntax. (The ANSI/ISO standard for the SQL language does not specify any syntax for construction of a database, the process by which a database comes into existence and has its name declared.)

Important: If you plan to use UTF-8 character encoding, including the Chinese GB18030-2000 code set, you must set the GL_USEGLU environment variable before you create the database.

The *database* that CREATE DATABASE specifies becomes the current database.

You cannot create a tenant database with the CREATE DATABASE statement. To create a tenant database, run the admin() or task() SQL administration API command with the tenant create argument.

If the DBCREATE_PERMISSION configuration parameter is not set, any user can create a database. If the configuration file includes one or more DBCREATE_PERMISSION specifications, however, only the specified users can create databases. Whether DBCREATE_PERMISSION is set, user **informix** can use the CREATE DATABASE statement. For additional information about how to set the DBCREATE_PERMISSION parameter to control which users can create new databases.

The *database* name that you declare must be unique within the database server environment in which you are working. The database server creates the system catalog tables that describe the structure of the new database.

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if a database of the specified name exists among the databases that are managed by the database server instance to which you are connected.

When you create a database, you alone can access it. It remains inaccessible to other users until you, as DBA, grant database privileges by running the GRANT statement.

If a previous CONNECT statement established an explicit connection to a database, and that connection is still your current connection, you cannot use the CREATE DATABASE statement (nor any SQL statement that creates an implicit connection) until after you use the DISCONNECT statement to close the explicit connection.

In , the CREATE DATABASE statement cannot appear in a multistatement PREPARE operation.

The SQL_LOGICAL_CHAR configuration parameter setting for the database server instance to which you are connected is recorded in the system catalog of the new database. The SQL_LOGICAL_CHAR configuration parameter controls the

expansion of size specifications in declarations of built-in character data types. This setting cannot be changed, and persists until the database is dropped, even if the Informix® instance that manages the database is stopped and restarted with a new SQL_LOGICAL_CHAR value. The **flags** column of the **systables** system catalog table encodes the SQL_LOGICAL_CHAR setting for the database.

If you do not specify a *dbspace*, the database server creates the system catalog tables in the **root** dbspace by default. However, if you enabled the automatic location of databases, databases are created by default in the dbspace that is chosen by the server. To enable the automatic location of databases, set the AUTOLOCATE configuration parameter or session environment variable to a positive integer.

The following statement creates the **vehicles** database in the **root** dbspace or in a dbspace that is chosen by the server, depending on whether automatic location is enabled:

```
CREATE DATABASE vehicles;
```

Because the example above includes no logging specification and no NLSCASE specification, then by default

- the **vehicles** database cannot support transaction logging,
- and if its locale uses a code set that distinguishes between uppercase and lowercase letters, the database is case-sensitive for all built-in character data types.

The following statement defines the **vehicles** database in the **research** dbspace:

```
CREATE DATABASE vehicles IN research;
```

But if no DROP DATABASE statement dropped existing the **vehicles** database that the first example created, the second example fails with an error, and no new database is created, because the identifier of the **vehicles** database is not unique for the database server instance.

Logging Options

The logging options of the CREATE DATABASE statement determine the type of transaction logging for the database. In the event of a failure, the database server uses the log to re-create all committed transactions in your database, unless CREATE DATABASE statement does not include the `WITH LOG` keywords.

To create a database that supports transaction logging, the CREATE DATABASE statement provides three syntax options:

- WITH BUFFERED LOG

  This option records transaction log information in a shared-memory buffer. This can allow the current transaction to complete sooner than in unbuffered logging mode, which requires that all the log information must be written to disk before the transaction can be completed. For a database using buffered logging, the database server does not write the contents of the logical-log buffer from shared memory to the logical log on disk until one of these events occurs:
  - The buffer becomes full,
  - or the connection is closed,

    - or a checkpoint occurs,
    - or a commit on a another database with unbuffered logging closes a distributed transaction, thereby flushing the buffer to disk.

  The following example uses the `WITH BUFFERED LOG` option to create a database with buffered logging:

  ```
  CREATE DATABASE bufDatabase WITH BUFFERED LOG;
  ```

- WITH LOG

  This option writes the log information directly to permanent storage, to reduce the risk of losing data if the server crashes while data rows are being modified. The following statement uses the `WITH LOG` keywords to create a database with unbuffered logging:

  ```
  CREATE DATABASE unbufDatabase WITH LOG;
  ```

- WITH LOG MODE ANSI

  This option enables implicit transactions. This mode also imposes certain requirements of the ANSI/ISO standard for the SQL language that the database server does not enforce in unlogged databases, or in databases that use explicit transactions.

  The following statement uses the `WITH LOG MODE ANSI` keywords to create an ANSI-compliant database with implicit transaction logging:

  ```
  CREATE DATABASE ansiDatabase WITH LOG MODE ANSI;
  ```

Unlogged databases

If you do not specify the `WITH LOG` keywords Informix® creates an unlogged database that cannot use transactions, and cannot run these SQL statements that support transaction logging:

- BEGIN WORK
- COMMIT WORK
- ROLLBACK WORK
- RELEASE SAVEPOINT
- ROLLBACK TO SAVEPOINT
- SET IMPLICIT TRANSACTION
- SET LOG
- SET ISOLATION.

The following example creates an unlogged database:

```
CREATE DATABASE unlogDatabase;
```

An unlogged database can improve performance in contexts where the cost of lost data is small. Alternatively, to move existing rows outside a transaction, you can create a database that supports transaction logging, but use RAW tables or light appends for unlogged operations on data that can be recovered from backup sources, if for some reason your load or unload operation fails.

Restrictions on logging modes

Some distributed operations require participating databases to have the same logging mode as the database from which the operation is initiated.

You must use the `WITH LOG` option when you create a database on a secondary server in a high-availability cluster.

Specifying Buffered Logging

The following example creates a database that uses a buffered log:

```
CREATE DATABASE vehicles WITH BUFFERED LOG;
```

If transactions are made against a database that uses buffered logging, the transaction log records are held in a logical-log memory buffer for as long as possible. They are not flushed from this logical-log buffer in shared memory to the logical log on disk until one of the following situations occurs:

- The buffer is full.
- A transaction is committed on the database with unbuffered logging.
- A checkpoint occurs.
- The connection is closed.

If you use a buffered log, you marginally enhance the performance of logging at the risk of not being able to re-create the last few transactions after an assertion failure.

ANSI/ISO-compliant databases cannot support buffered logging, but must be created with the `LOG MODE ANSI` keywords:

```
CREATE IF NOT EXISTS DATABASE nobuficles WITH LOG MODE ANSI;
```

See also the discussion of buffered logging in the *IBM® Informix® Database Design and Implementation Guide*.

ANSI-Compliant Databases

When you use the LOG MODE ANSI option in the CREATE DATABASE statement, the database that you create is an ANSI-compliant database that conforms to the ANSI/ISO standard for the SQL language.

The following example creates an ANSI-compliant database:

```
CREATE DATABASE employees WITH LOG MODE ANSI;
```

ANSI-compliant databases are different from databases that are not ANSI compliant in several ways, including the following features:

- All SQL statements are automatically contained in transactions.
- All databases use unbuffered logging.
- Owner naming is enforced.

  You must qualify with the owner name any table, view, synonym, index, or constraint that you do not own. Unless you enclose the owner name between quotation marks, alphabetic characters in owner names default to uppercase. (To

prevent this upshifting of lowercase letters in undelimited owner names, you can set the **ANSIOWNER** environment variable to `1`.)

In addition, the routine signature of a UDR includes the name of the owner; in databases that are not ANSI compliant, this is true only for the **sysdbopen( )** and **sysdbclose( )** procedures.

- For sessions, the default isolation level is REPEATABLE READ.
- Default privileges on objects differ from those in databases that are not ANSI compliant. When you create a table or a synonym, other users do not receive access privileges (as members if the PUBLIC group) on the object by default.
- All DECIMAL data types are fixed-point values. If you declare a column as DECIMAL(*p*), the default scale is zero, meaning that only integer values can be stored. (In a database that is not ANSI compliant, DECIMAL(*p*) is a floating-point data type of a scale large enough to store the exponential notation for a value.)

Other slight differences exist between databases that are ANSI compliant and those that are not. These differences are noted with the related SQL statement in this document. For a detailed discussion of the differences between ANSI compliant databases and databases that are not ANSI-compliant, see the *IBM® Informix® Database Design and Implementation Guide*.

Creating an ANSI-compliant database does not mean that you automatically receive warnings for Informix® extensions to the ANSI/ISO standard for SQL syntax when you run the database. You must also use the **-ansi** flag or the **DBANSIWARN** environment variable to receive such warnings.

For additional information about **-ansi** and **DBANSIWARN**, see the *HCL® Informix® Guide to SQL: Reference*.

Specifying NLSCASE case sensitivity

You can explicitly create a case-sensitive or case insensitive database.

By default, in databases where the locale classifies disjunct subsets of the code set as *uppercase letters* and as *lowercase letters*, Informix® databases are created as case-sensitive. The database locale is defined by setting the **DB_LOCALE** environment variable. An example of a locale whose code set recognizes letter case is the default US English locale, where lowercase letters precede uppercase letters in the ascending order of collation. In the default locale, the following statement creates a case-sensitive database:

```
CREATE DATABASE employees IN dbspaceYee WITH BUFFERED LOG;
```

To explicitly create a case-sensitive database, include the `NLSCASE SENSITIVE` keywords as the last specification of the CREATE DATABASE statement, as in the following example:

```
CREATE DATABASE stores IN dbsp1 WITH LOG NLSCASE SENSITIVE;
```

Because case sensitivity is enabled by default, the following statement has the same effect:

```
CREATE DATABASE stores IN dbsp1 WITH LOG;
```

In a database that is case sensitive, for a table in which column **col3** is of type NCHAR or NVARCHAR, for example, the Boolean condition `col3 MATCHES 'SAM'` evaluates as false for rows where `'Sam'` is the value in **col3**. In contrast, in a different database created with the `NLSCASE INSENSITIVE` keyword option, the same Boolean condition `col3 MATCHES 'SAM'` evaluates as true for the same rows of the same table. As this example implies, for tables that include NCHAR or NVARCHAR columns in which some row values differ only in letter case, query results depend on the `NLSCASE` setting of the database.

All informix databases are case sensitive for operations on character strings of the built-in CHAR, LVARCHAR, and VARCHAR data types. If you create a case-sensitive database, whether by default, or explicitly with the `NLSCASE SENSITIVE` keywords, that database also treats strings of the National Language Support data types NCHAR and NVARCHAR as case-sensitive, if the database locale supports letter case.

Creating a database that is not case sensitive

In some applications, the letter case of character strings can be disregarded. Data entry procedures, for example, might accept the strings `'M'` and `'m'` as logically equivalent within a record. For large data sets, applying conditional logic to convert both case variants to a single value can result in slower performance than storing the records in an NCHAR or NVARCHAR column of a case-insensitive database, in which the strings `'M'` and `'m'` encode the same case-insensitive value. Here the condition `'M' MATCHES 'm'` evaluates as true for NCHAR or NVARCHAR columns.

Every database created with the `NLSCASE INSENSITIVE` property stores uppercase and lowercase NCHAR and NVARCHAR letters exactly as they are loaded into their tables; any unmodified record that a query returns has its original lettercase. In all operations on NCHAR and NVARCHAR values, however, such as sorting, grouping, or identifying duplicate rows, the database server ignores any letter case variants, and treats, for example, the strings `'Mi'` and `'mI'` as the same value. Information about the case of letters not discarded, but it is also not used when the database server processes NLS data types.

When you include the `NLSCASE INSENSITIVE` keywords as the last specification of the CREATE DATABASE statement, the database server creates a database that always processes the following types of character strings without regard to letter case:

- Strings stored in columns of the NLS data types NCHAR and NVARCHAR
- Strings stored as a DISTINCT data type whose base type is NCHAR or NVARCHAR
- Strings stored as elements of these data types within a collection data type
- Strings stored in fields of the above data types in a named or unnamed ROW data type
- Strings stored as SPL variables of these data types
- Strings implicitly or explicitly cast to these data types
- Strings returned by functions as output parameters of these data types.

Here "*these data types*" refers to the character data types that are identified in the same list.

The following statement creates a database with the `NLSCASE` property set to `INSENSITIVE`:

```
CREATE DATABASE stores IN dbsp2 WITH BUFFERED LOG NLSCASE INSENSITIVE;
```

Important: A database created as `NLSCASE INSENSITIVE` treats all other built-in character data types (CHAR, LVARCHAR, and VARCHAR) as case sensitive. That is, a case-insensitive database can also perform case-sensitive processing of string values, if their data types are not among the NLS character data types in the list above.

To perform case-sensitive operations on a string of the NCHAR or NVARCHAR data type in a case-insensitive database, you must first explicitly cast the string to a CHAR, LVARCHAR, or VARCHAR data type, and then perform the case-sensitive operation. (See, however, the topic Return Types from CONCAT and String Functions on page 1115, which identifies contexts

in which the database server automatically casts the result of built-in string-manipulation functions and string operators to NCHAR or NVARCHAR data types.)

Examples of NLSCASE INSENSITIVE queries

In a case-insensitive database, when a query calls an aggregate function or includes the GROUP BY clause for an NCHAR or NVARCHAR column, the database server treats letter-case variants in the data as duplicate column values, as in the following program fragment.

```
CREATE DATABASE casedb WITH LOG NLSCASE INSENSITIVE;
CREATE TABLE foo (cc CHAR(5), nc NCHAR(5));
INSERT INTO foo VALUES ('IBM', 'iBM');
INSERT INTO foo VALUES ('ibm', 'ibM');
INSERT INTO foo VALUES ('ibm', 'ibM');
INSERT INTO foo VALUES ('Ibm', 'Ibm');

SELECT COUNT(nc) FROM foo
   GROUP BY nc;
SELECT COUNT(nc) FROM foo
   WHERE nc = 'ibm' GROUP BY nc;
```

In both of the queries above, the **COUNT** aggregate function returns `4`, the total number of rows that the INSERT statements loaded into **foo**. Because column **nc** is an NLS data type, all of the rows satisfy the `nc = 'ibm'` condition in the WHERE clause, despite variations in letter case among the **nc** values.

For the following query on the same tables,

```
SELECT nc FROM foo GROUP BY nc;
```

the output can be any of the string values from the INSERT statements (namely `'IBM'`, `'iBM'`, `'ibm'`, `'ibM'`, or `'Ibm'`), depending on the order in which the server processes or scans the rows.

The next query on the same table excludes duplicate rows from the result set by including the DISTINCT keyword in the projection clause:

```
SELECT DISTINCT nc FROM foo;
```

This returns a single row, because from the **`NLSCASE INSENSITIVE`** perspective, all of the rows have the same value, despite the variations in letter case. As in the previous example, the first row retrieved from among the inserted rows will be returned by the query.

The next example includes the DISTINCT keyword as one of the arguments to the **COUNT** aggregate function:

```
SELECT COUNT(DISTINCT nc) FROM foo;
```

This returns a count of `1`, again because in this case-insensitive database, all of the rows in table **foo** evaluate as duplicates.

Restrictions on NLSCASE INSENSITIVE databases

The following restrictions apply to databases that are created with the NLSCASE INSENSITIVE property:

- They support distributed cross-database and cross-server queries only with databases that also have the NLSCASE INSENSITIVE property.
- Case-sensitive databases cannot connect to NLSCASE INSENSITIVE databases. Attempts to do so fail with this error:

```
-26801 Cannot reference an external database that is not case sensitive.
```

- NLSCASE INSENSITIVE databases cannot connect to case-sensitive databases. Attempts to do so fail with this error:

```
-26802 Cannot reference an external database that is case sensitive.
```

  The only exception is that the NLSCASE setting does not prevent connections to the case-sensitive system databases, such as **sysmaster**, **sysadmin**, **sysutils**, **sysusers**, and **syscdr**, of the same Informix® database server instance. The results of operations that access a system database depends on the NLSCASE setting of the other database.
- The **onload** and **onunload** utilities do not support databases that have the NLSCASE INSENSITIVE property.
- In an Enterprise Replication cluster, no error or warning is issued if you specify a replication pair in whose databases differ in their NLSCASE property. To reduce the risk of data inconsistencies, replicate case-sensitive databases only with case-sensitive databases. and replicate NLSCASE INSENSITIVE databases only with NLSCASE INSENSITIVE databases.

CREATE DEFAULT USER statement (UNIX™, Linux™)

Use the CREATE DEFAULT USER statement to define the properties set of the default internally authenticated user. This statement is an extension to the ANSI/ISO standard for the SQL language.

Syntax



Usage

CREATE DEFAULT USER is a special case of the CREATE USER statement. After you use the CREATE DEFAULT USER statement to define default user properties, you can use the CREATE USER statement (but omitting the PROPERTIES clause) to create new users who have default user properties.

Only a DBSA can issue the CREATE DEFAULT USER statement. With a non-root installation, the user who installs the server is the equivalent of the DBSA, unless the user delegates DBSA privileges to a different user.

The USERMAPPING configuration parameter must be set to a value (ADMIN or BASIC) that enables support for mapped users before default users who were created with the CREATE DEFAULT USER statement can connect to the database server. A DBSA can issue the CREATE DEFAULT USER statement to map default users to properties that correspond to the appropriate level of authorization. The USERMAPPING configuration parameter must be set to ADMIN to enable a default user to have a server administrative privilege with the AUTHORIZATION keyword, where `AAO`, `BARGROUP`, `DBSA`, and `DBSSO` are the keyword options for specific administrative privileges.

You must also enter values in the SYSUSERMAP table of the **sysusers** database to map users with the appropriate user properties, so that the mapped user statements of SQL can work correctly.

You cannot specify a password, or account lock, or account unlock information in the CREATE DEFAULT USER statement. This statement is equivalent to the `GRANT ACCESS TO PUBLIC PROPERTIES` statement. The equivalent syntax to

```
REVOKE ACCESS TO PUBLIC;
```

is this:

```
DROP DEFAULT USER;
```

To alter the properties of the default internally authenticated user, you can issue the `ALTER DEFAULT USER WITH PROPERTIES` statement.

Execution of the CREATE DEFAULT USER statement can be audited with the CRUR audit code, which is the same mnemonic as for the CREATE USER statement.

For more information about the PROPERTIES options to the CREATE DEFAULT USER statement, see the CREATE USER statement (UNIX, Linux) on page 416.

CREATE DISTINCT TYPE statement

Use the CREATE DISTINCT TYPE statement to create a new distinct data type.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
CREATE DISTINCT TYPE [IF NOT EXISTS] distinct_type assource_type
```

Element	Description	Restrictions	Syntax
distinct_type	Name that you declare here for the new distinct data type	In an ANSI-compliant database, the combination of the owner and data type must be unique within the database. In a database that is not ANSI compliant, the name must be unique among names of data types in the database.	Data Type on page 985
source_type	Name of existing type on which the new type is based	Must be either a built-in data type or one created with the CREATE DISTINCT TYPE, CREATE OPAQUE TYPE, or CREATE ROW TYPE statement	Data Type on page 985

Usage

A *distinct type* is a data type based on a built-in data type or on an existing opaque data type, a named ROW data type, or another distinct data type. Distinct data types are strongly typed. Although the distinct type has the same physical representation of data as its source type, values of the two types cannot be compared without an explicit cast from one type to the other.

To create a distinct data type, you must have the Resource privilege on the database. Any user with the Resource privilege can create a distinct type from one of the built-in data types, which user **informix** owns.

Important: You cannot create a distinct type on the SERIAL, BIGSERIAL, or SERIAL8 data types.

To create a distinct type from an opaque type, from a named ROW type, or from another distinct type, you must be the owner of the data type or have the Usage privilege on the data type.

By default, after a distinct type is defined, only the owner of the distinct type and the DBA can use it. The owner of the distinct type, however, can grant to other users the Usage privilege on the distinct type.

A distinct type has the same storage structure as its source type. The following statement creates the distinct type **birthday**, based on the built-in DATE data type:

```
CREATE DISTINCT TYPE birthday AS DATE;
```

Although Informix® uses the same storage format for the distinct type as it does for its source type, a distinct type and its source type cannot be compared in an operation unless one type is cast explicitly to the other type.

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if a DISTINCT data type of the specified name is already registered in the current database.

Privileges on Distinct Types

To create a distinct type, you must have the Resource privilege on the database. When you create the distinct type, only you, the owner, have Usage privilege on this type. Use the GRANT or REVOKE statements to grant or revoke Usage privilege to other database users.

To find out what privileges exist on a particular type, check the **sysxtdtypes** system catalog table for the owner name and the **sysxtdtypeauth** system catalog table for additional data type privileges that might have been granted. For more information on system catalog tables, see the *HCL® Informix® Guide to SQL: Reference*.

The utility can also display privileges on distinct types.

Support Functions and Casts

When you create a distinct type, Informix® automatically defines two explicit casts:

- A cast from the distinct type to its source type
- A cast from the source type to the distinct type

Because the two data types have the same representation (the same length and alignment), no support functions are required to implement the casts.

You can create an implicit cast between a distinct type and its source type. To create an implicit cast, use the Table Options clause to specify the format of the external data. You must first drop the default explicit cast, however, between the distinct type and its source type.

All support functions and casts that are defined on the source type can be used on the distinct type. Casts and support functions that are defined on the distinct type, however, are not available to the source type. Use the Table Options clause to specify the format of the external data.

Manipulating Distinct Types

When you compare or manipulate data of a distinct type and its source type, you must explicitly cast one type to the other in the following situations:

- To insert or update a column of one type with values of the other type
- To use a relational operator to add, subtract, multiply, divide, compare, or otherwise manipulate two values, one of the source type and one of the distinct type

For example, suppose you create a distinct type, **dist_type**, that is based on the NUMERIC data type. You then create a table with two columns, one of type **dist_type** and one of type NUMERIC.

```
CREATE DISTINCT TYPE dist_type AS NUMERIC;
CREATE TABLE t(col1 dist_type, col2 NUMERIC);
```

To directly compare the distinct type and its source type or assign a value of the source type to a column of the distinct type, you must cast one type to the other, as the following examples show:

```
INSERT INTO tab (col1) VALUES (3.5::dist_type);

SELECT col1, col2
   FROM t WHERE (col1::NUMERIC) > col2;

SELECT col1, col2, (col1 + col2::dist_type) sum_col
   FROM tab;
```

For information about queries and other distributed DML operations that access DISTINCT data types in tables outside the local database, see the topic DISTINCT Types in Distributed Operations on page 996.

CREATE EXTERNAL TABLE Statement

Use the CREATE EXTERNAL TABLE statement to define an external source that is not part of your database to load and unload data for your database.

The implementation of the CREATE EXTERNAL TABLE statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_274_Connect_42_crext001) unique_274_Connect_42_crext001(explicit id unique_274_Connect_42_crext002)

unique_274_Connect_42_crext002(explicit id unique_274_Connect_42_crext003) unique_274_Connect_42_crext003

```
CREATE EXTERNAL TABLE [IF NOT EXISTS] table <Column Definition>[] USING( [ <Table Options> [] ] <DATAFILES Clause>[] [ <Table Options> [] ] )
```

Element	Description	Restrictions	Syntax
table	The name of the table to store external data	Must be unique among names of tables, views, and synonyms in the current database	Identifier on page 1231

Usage

You use external tables to load and unload data to or from your database. You can also use external tables to query data in text files that are not in Informix® databases.

The first portion of the syntax diagram declares the name of the table and defines its columns.

The portion that follows the USING keyword identifies external files that the database server opens when you use the external table, and specifies additional options for characteristics of the external table.

After executing the CREATE EXTERNAL TABLE statement, you can move data to and from the external source with an INSERT INTO ... SELECT statement. See the section INTO EXTERNAL clause for more information about loading the results of a query into an external table.

The CREATE EXTERNAL TABLE statement is not supported on secondary servers within a high-availability cluster.

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if an external table of the specified name is already registered in the **systables** system catalog table of the current database, or if the specified name is the identifier of a database table, view, sequence object, or synonym in the current database.

DATAFILES Clause

The DATAFILES clause specifies the operating system file or pipe that is opened when you use an external table.

DATAFILES Clause

```
"DATAFILES"
"('{DISK | PIPE}:{fixed_path | formatted_path}[;{BLOBDIR | CLOBDIR}:fixed_path]')"
```

Element	Description	Restrictions	Syntax
fixed_path	Path name for input or output files in the definition of the external table	See the notes that follow this table	Must conform to operating-system rules
formatted_path	Formatted path name that uses pattern-matching characters	See the notes that follow this table	Must conform to operating-system rules

The database server does not verify that any file or pipe exists at the specified *fixed_path* or *formatted_path*, that the specified pipe is open, nor that the user has permission to access that file system. Subsequent operations on the external table will fail, however, unless the path is valid and, if a named pipe is being used, that it is open, when the database server attempts to read or write to the external table.

For examples of the DATAFILES clause, see External Table Examples on page 210.

Keyword

Description

CLOBDIR

Specifies the server directory in which the CLOB file is stored.

BLOBDIR

Specifies the server directory in which the BLOB file is stored. When creating queries, specify DISK followed by BLOBDIR followed by CLOBDIR. If BLOBDIR is omitted, BLOB files are stored the same directory as specified by the DISK clause. If both BLOBDIR and CLOBDIR are omitted, a new file is created for each BLOB or CLOB column and stored in the directory in which the DISK clause is specified.

In the following example, rows stored in `/work1/exttab1.dat` have their BLOBs located in `/work1/blobdir1` and CLOBs in the `/work1/clobdir1` directory.

Rows stored in `/work1/exttab2.dat` have their BLOBs located in the `/work1` directory and CLOBs in the `/work1/clobdir2` directory. Because the BLOBDIR clause is omitted, the BLOBs are stored in the directory where `exttab2.dat` is stored.

Rows stored in the `/work1/exttab3.dat` have their BLOBs and CLOBs located in the `/work1` directory because both BLOBDIR and CLOBDIR are omitted.

```
CREATE EXTERNAL TABLE exttab (
       id    SERIAL,
       lobc  CLOB,
       lobb  BLOB)
USING (DATAFILES(
 "DISK:/work1/exttab1.dat;BLOBDIR:/work1/blobdir1;CLOBDIR:/work1/clobdir1",
 "DISK:/work1/exttab2.dat;CLOBDIR:/work1/clobdir2",
 "DISK:/work1/exttab3.dat"),
 DELIMITER '|');
```

Using Formatting Characters with External Tables

You can use a formatted path name to designate a file name by using the substitution character `%r` (*first ..last*).

Formatting String

Effect

%r(*first ..last*)

Specifies multiple files on the Informix® server for the external table.

The *first* and *last* arguments represent a range of values that are substituted in the expression when the statement is run. For example, specifying my_file.%r(1..3) expands to:

my_file.1

my_file.2

my_file.3

The only supported formatting character supported by Informix® is `%r`.

Table options

These options specify additional characteristics that define the external table, and that define attributes of load or unload operations on that table.

(explicit id unique_276_Connect_42_mod002a) unique_276_Connect_42_mod002a(explicit id unique_276_Connect_42_mod002b)

unique_276_Connect_42_mod002b

Table Options

" { | **FORMAT** [' { **DELIMITED** | **INFORMIX** | **FIXED** } '] | **DEFAULT** | <Loading mode options> [ ] | { **DBDATE***'date_format'* | **DBMONEY***'currency'* | **DELIMITER***'field_delimiter'* | **RECORDEND***'record_delimiter'* | **MAXERRORS***num_errors* | **REJECTFILE***'filename'* | [**ESCAPE** { **ON** | **[OFF]** }] | { **NUMROWS** | **SIZE** } *num_rows* } } "

Loading mode options

" { [ ] | **EXPRESS** | **DELUXE** } "

Element	Description	Restrictions	Syntax
field_delimiter	Character that separates fields. Default is pipe (|) character	For nonprinting characters, use octal notation.	Quoted String on page 1202
filename	Full path name for conversion error messages	See Reject Files on page 208.	Must conform to operating-system rules.
num_errors	Number of errors before load operations are terminated	Value is ignored unless the REJECTFILE value is set. This specification is ignored during unload tasks.	Literal Number on page 1198
num_rows	Approximate number of rows contained in the external table	Must be a positive number.	Literal Number on page 1198

Element	Description	Restrictions	Syntax
record_delimiter	Character to separate records. Default is Newline (`\n`)	For nonprinting characters, use octal.	Quoted String on page 1202

Usage

If no RECORDEND value is specified, *record_delimiter* defaults to the Newline character (`\n`). To specify a nonprinting character as the record delimiter or field delimiter, you must encode it as the octal representation of the ASCII character. For example, `\006` can represent CTRL-F.

On Windows™ systems, if you use the DB-Access utility or the dbexport utility to unload a database table into a file and then plan to use the file as an external table datafile, you should define RECORDEND as '\012' in the CREATE EXTERNAL TABLE statement.

Use the table options keywords as the following table describes. You can use each keyword whenever you plan to either load or unload data except where noted.

Keyword

Description

DBDATE

Specifies the date format when reading or writing an external table. You use the DBDATE clause to convert data during load and unload operations from external tables. In the following example, DBDATE is set to `DMY2-`. If the date value in the database table was stored as 06/24/2009, the value written to the external table is 24-06-09.

```
CREATE EXTERNAL TABLE ext_date (dob date)
USING (  DATAFILES ("DISK:/tmp/datedisk"),
       REJECTFILE "/tmp/datereject",
       DBDATE "DMY2-",
       FORMAT "delimited");

INSERT INTO ext_date SELECT * FROM basetab;
```

The DBDATE clause is also used when inserting date values from external tables into database tables. In the following example, data in the external table is converted to internal binary format based on the DBDATE value set by the CREATE EXTERNAL TABLE statement.

```
INSERT INTO basetab SELECT * FROM ext_date;
```

If the DBDATE keyword is not specified in the USING clause of the CREATE EXTERNAL TABLE statement, the date format is determined by the setting of the DBDATE environment variable. If the DBDATE environment variable is not specified, the date format is determined by the setting of the GL_DATE environment variable. The value specified by the DBDATE clause takes precedence over the value specified by the DBDATE environment variable. The setting of the DBDATE variable takes precedence over that of the GL_DATE environment variable. See the *HCL® Informix® Guide to SQL: Reference* for information about DBDATE and GL_DATE values.

DBMONEY

Specifies the currency format when reading or writing an external table. You use the DBMONEY clause to convert data during load and unload operations from external tables. In the following example, DBMONEY is set to `DM,` . Currency is formatted as DM (deutsche mark) units, using the currency symbol `DM` and comma (,) . If the currency value in the database table is stored as `100.50`, the value written to the external table is `100,50`.

```
CREATE EXTERNAL TABLE ext_money (sales money)
USING ( DATAFILES ( "DISK:/tmp/moneydisk" ),
      REJECTFILE "/tmp/moneyreject",
       DBMONEY "DM,",
       FORMAT "delimited");

INSERT INTO ext_money SELECT * FROM basetab;
```

When reading data from an external table into a database table, the currency symbol is not required in the external table. For example, if the external table contained the value `1000,78` and DBMONEY was set to `DM,` then the data is not rejected and the row is stored correctly.

If the decimal separator in the external table and the value set for DBMONEY do not match, then the row is rejected. For example, if the external table contained the value `1000,78` (with a comma instead of a decimal point) and the DBMONEY clause is set to `DM.` then the row is rejected. If the data file contains a currency symbol and the currency symbol does not match the DBMONEY currency symbol, the row is rejected.

When writing data from a database table into an external table, the currency symbol is not written to the external table.

If the DBMONEY clause is not specified, the data format is determined by the setting of the DBMONEY environment variable. The value specified by the DBMONEY clause takes precedence over the value specified by the DBMONEY environment variable. If the DBMONEY clause is not specified and the DBMONEY environment variable is not set, the decimal separator specified by the database locale is used. See the *HCL® Informix® Guide to SQL: Reference* for information about DBMONEY values.

DEFAULT (load only)

Specifies replacing missing values in delimited input files with column defaults (if they are defined) instead of NULLs, so input files can be sparsely populated. Files do not need an entry for every column in the file where a default is the value to be loaded.

DELIMITED

Specifies that the data file is a delimited text file. A delimiter character can be specified using the optional DELIMITER table option.

DELIMITER

Specifies the character that separates fields in a delimited text file. If the table options include no DELIMITER specification, the pipe (`|`) character is the default field separator.

DELUXE (load only)

The database server always uses DELUXE mode for STANDARD tables if the database uses transaction logging, and on any table on which an index is defined.

You can specify DELUXE mode to override the default EXPRESS load mode for RAW target tables without indexes if the database is logged.

The DELUXE mode updates indexes, performs constraint checking, and evaluates triggers as data is inserted into the table. DELUXE mode loads are not as fast as EXPRESS mode loads, but are more flexible. In DELUXE mode, you can access and update the table that is being loaded.

ESCAPE

Inserts the default escape character immediately before any instances of the *field_delimiter* separator that DELIMITER specifies, where that character is a literal value in the data, rather than a separator. Whether you include or omit the ESCAPE keyword, this functionality is enabled by default, or you can specify the `ESCAPE ON` keywords to make it clearer to human readers of your SQL code that this feature is enabled. To prevent literal *field_delimiter* separator characters in the data from being escaped, you must specify the `ESCAPE OFF` keywords.

By default, the escape character that the ESCAPE keyword inserts before literal *field_delimiter* characters is the backslash (`\`) character. But if the DEFAULTESCCHAR configuration parameter is set to a single-character value, that character replaces backslash (`\`) for delimiter characters used as literals when `ESCAPE` or `ESCAPE ON` is specified.

 **Note:**

The default setting for ESCAPE is OFF in Informix® releases earlier than version 12.10.

EXPRESS (load only)

EXPRESS mode is always used if the database is not logged and the target table (of any table type) has no indexes.

EXPRESS is the default if the database is logged for RAW target tables without indexes; however, you can override that default by specifying the DELUXE keyword.

EXPRESS mode loads use light appends and are significantly faster than DELUXE mode loads, but less flexible. In EXPRESS mode you cannot update the table or read the new data entries until the load is complete.

An error message is generated and the load is stopped if EXPRESS mode is specified and the table contains objects of type BLOB, BYTE, CLOB, or TEXT.

When data is loaded using EXPRESS mode, the target table cannot be located within an Enterprise Replication (ER) replicate. In addition, the target database server must not have high-availability data replication (HDR) enabled.

FIXED

Specifies that the data file is fixed width. When using EXTERNAL data types in the external table, the FIXED format must be used.

FORMAT

Specifies the format of the data in the data files.

INFORMIX

Specifies that the format of the data file is internal Informix® format. Loading data from an external table saved in Informix® format is faster than loading data from a fixed or delimited external file. Use Informix® format when moving data from one Informix® database to another.

MAXERRORS

Sets the number of errors that are allowed before the database server stops loading data.

The minimum value for MAXERRORS is 1. Setting MAXERRORS to a value less than 1 produces an error. The maximum value for MAXERRORS is 2,147,483,647.

RECORDEND

Specifies the character that separates records in a delimited text file.

REJECTFILE

Sets the full path name where the database server writes data-conversion errors. If not specified or if files cannot be opened, any error ends the loading of data abnormally. See also Reject Files on page 208.

NUMROWS *or* SIZE

The approximate number of rows in the external table.

Specifying NUMROWS (or its synonym, SIZE) can improve performance when an external table is used in a join query. This value cannot be NULL.

Reject Files

Rows that have conversion errors during a load or rows that violate check constraints on the external table are written to a reject file. The REJECTFILE clause declares the path and file name of the reject file.

If you perform another load to the same table during the same session, any earlier reject file of the same name is overwritten.

Reject file entries have the following format:

```
filename, record, reason-code,
  field-name: bad-line
```

The following table describes these elements of the reject file:

Element

Description

filename

Name of the input file.

record

Record number in the input file where the error was detected.

reason-code

Description of the error.

field-name

External field name where the first error in the line occurred, or `<none>` if the rejection is not specific to a particular column.

bad-line

Line that caused the error (delimited or fixed-position character files only).

The reject file writes the *filename, record*, *field-name,* and *reason-code* in ASCII. The *bad-line* information varies with the type of input file.

- For delimited files or fixed-position character files, up to 80 characters of the bad line are copied directly into the reject file.
- For Informix® internal data files, the bad line information is not placed in the reject file because you cannot edit the binary representation in a file; but the *filename, record, reason-code*, and *field-name* are still reported in the reject file so you can isolate the problem. Use the Table Options clause to specify the format of the external data.

The following errors can cause a row to be rejected.

Error Text

Explanation

CONSTRAINT *constraint name*

This constraint was violated.

CONVERT_ERR

Any field encounters a conversion error.

MISSING_DELIMITER

No delimiter was found.

MISSING_RECORDEND

No end of record was found.

NOT NULL

A NULL was found in *field-name*.

ROW_TOO_LONG

The input record is longer than 32 kilobytes.

Virtual Processors

A FIFO virtual processor is used by external tables in Informix®. One FIFO virtual processor is created when the server is initialized. Additional FIFO virtual processors can be added using the onmode -p command. For example, use the following command to add three FIFO virtual processors:

```
onmode -p +3 fifo
```

FIFO virtual processors cannot be deleted.

The FIFO virtual processors are used to process I/O related to the pipes that are defined with the PIPE clause.

See the *HCL® Informix® Administrator's Guide* for more information about using FIFO virtual processors.

External Table Examples

The examples in this section illustrate different ways to load and unload data using external tables.

Example

The following is an example of the CREATE EXTERNAL TABLE syntax. In the example, an external table named `empdata` is created with two columns. The DATAFILES clause indicates the location of the data file, specifies that the file is delimited, indicates the location of the reject file, and indicates that the reject file can contain no more than 100 errors.

```
CREATE EXTERNAL TABLE empdata
(
   empname char(40),
   empdoj date
)
USING
(DATAFILES
    (
        "DISK:/work/empdata.unl"
    ),
    FORMAT "DELIMITED",
    REJECTFILE "/work/errlog/empdata.rej",
    MAXERRORS 100);
```

Example

Creating an external table using the SAMEAS clause

The SAMEAS *template* clause uses all the column names and data types from the *template* table in the definition of the new table. The following example uses the column names and data types of the `empdata` table and uses them for the external table.

```
CREATE EXTERNAL TABLE emp_ext SAMEAS empdata
USING
(DATAFILES
    (
        "DISK:/work/empdata2.unl"
    ),
    REJECTFILE "/work/errlog/empdata2.rej",
```

```
    DELUXE
);
```

Example

Unloading data into an external table

The following example shows statements used to load data from a database table into an external table.

```
CREATE EXTERNAL TABLE ext1( col1 int )
 USING
 (DATAFILES
     (
        "DISK:/tmp/ext1.unl"
     )
);

CREATE TABLE base (col1 int);
INSERT INTO ext1 SELECT * FROM base;
```

You can also use the SELECT...INTO EXTERNAL syntax to unload data as in the following example.

```
SELECT * FROM base
INTO EXTERNAL emp_target
 USING
 (DATAFILES
     (
          "DISK:/tmp/ext1.unl"
     )
);
```

Example

Selecting from an external table and loading into a database table

The following example selects from an external and shows various ways to load external data into a database table.

```
CREATE EXTERNAL TABLE ext1( col1 int )
  USING
  (DATAFILES
      (
          "DISK:/tmp/ext1.unl“
      )
);

CREATE TABLE target1 (col1 int);
CREATE TABLE target2 (col1 serial8, col2 int);

SELECT * FROM ext1;
SELECT col1,COUNT(*) FROM ext1 GROUP BY 1;
SELECT MAX(col1) FROM ext1;
SELECT col1 FROM ext1 a, systables b WHERE a.col1=b.tabid;

INSERT INTO target1 SELECT * FROM ext1;
INSERT INTO target2 SELECT 0,* FROM ext1;
```

Example

Unloading from a database table to a text file using FIXED format

The next example creates an external table named `emp_ext`, defines the column names and data types, and unloads the data from the database using fixed format.

```
CREATE EXTERNAL TABLE emp_ext
 ( name CHAR(18) EXTERNAL CHAR(20),
   address VARCHAR(40) EXTERNAL CHAR(40),
   empno INTEGER EXTERNAL CHAR(6)
  )
USING (
  FORMAT 'FIXED',
  DATAFILES
     (
        "DISK:/work2/mydir/emp.fix"
     )
);

INSERT INTO emp_ext SELECT * FROM employee;
```

Example

Loading data from a data file into a database table using FIXED format

The next example creates an external table named `emp_ext` and loads data into the database from a fixed format file.

```
CREATE EXTERNAL TABLE emp_ext
 ( name CHAR(18) EXTERNAL CHAR(18),
   address VARCHAR(40) EXTERNAL CHAR(40),
   empno INTEGER EXTERNAL CHAR(6)
 )
USING (
  FORMAT 'FIXED',
  DATAFILES
     (
         "DISK:/work2/mydir/emp.fix"
     )
);

INSERT INTO employee SELECT * FROM emp_ext;
```

Example

Using formatting characters in the DATAFILES clause

To process three files, create the DATAFILES clause as in the following example.

```
DATAFILES
   (
      "DISK:/work2/extern.dir/mytbl.%r(1..3)"
   )
```

The following shows how the list is expanded when the statement is run:

```
DATAFILES
   (
      "DISK:/work2/extern.dir/mytbl.1",
```

```
    "DISK:/work2/extern.dir/mytbl.2",
    "DISK:/work2/extern.dir/mytbl.3"
)
```

Loading Data from External Tables into Informix®

To load data, you define the external data as an external table and then insert the data into the database.

The database server performs express-mode loads and deluxe-mode loads. You can perform express-mode loads only when the table is type RAW and does not have any active indexes. The database server allows constraint checking for both load modes.

Express® mode provides the highest performance during a load.

Deluxe mode combines fast parallel loading with evaluation of indexes and unique constraints, and is more efficient in the following situations:

- The cost of rebuilding an index is too high for the amount of data that you are loading.
- You want to use the empty space from deleted rows in the table that you are loading. database

If the table receiving the rows from the external table is a STANDARD table (that is, a database table that was not created by the CREATE TEMP TABLE or CREATE RAW TABLE statement), the EXPRESS keyword has no effect, and the table is loaded in DELUXE mode. The database server does not issue an exception when it ignores the EXPRESS keyword in load operations where the receiving table is not a RAW table.

Loading Data in Express® Mode

Express® mode supports rapid loading of data into tables that have no indexes. In logged databases, only RAW tables can use this mode.

Before you begin

Warning: Express-mode loads are not allowed for STANDARD tables in databases that support transaction logging.

About this task

Express-mode loads use light appends, which bypass the buffer pool. Light appends eliminate the overhead associated with buffer management but do not log the data. In express mode, the database server automatically locks the table exclusively. No other users can access the table.

You can use express mode for any newly created table with no data if you define the table as type RAW and do not define any indexes until after you load the data. Choose RAW tables if you do not want to use logging in a database that supports transaction logging.

To prepare an existing table for express-mode load, drop all indexes, and make sure the table type is RAW.

Data loaded from an external table into a raw table is not logged; therefore, you must perform a level-0 backup before the database can be dropped. If you try to drop the database before you perform a level-0 backup, the database server issues ISAM error -197, as follows:

```
Partition recently appended to; can't open for write or logging
```

Consider a table with the following schema:

```
TABLE employee (
   name CHAR(18),
   hiredate DATE,
   address CHAR(40),
   empno INTEGER);
```

To use express-mode load on an existing table

1. Alter the table type to allow fast loading.

   ```
   ALTER TABLE employee TYPE (RAW);
   ```

2. Create the external table description.

   ```
   CREATE EXTERNAL TABLE emp_ext
   SAMEAS employee
   USING (
      FORMAT 'DELIMITED',
      DATAFILES
        ("DISK:/work2/mydir/emp.dat"),
      REJECTFILE "/work2/mydir/emp.rej",
      EXPRESS
      );
   ```

3. Load the table.

   ```
   INSERT INTO employee SELECT * FROM emp_ext;
   ```

   If the database server chooses express mode, the load stops with an error message if the destination table contains indexes, constraints, or any other problem conditions.

4. Create a level-0 backup.

   Because the data is not logged, you must perform a level-0 backup to allow data recovery. If a disk fails, you cannot recover the data automatically. You need to use the most recent level-0 backup files. ,

Results

If the table type is RAW (nonlogging), omit the statements BEGIN WORK and COMMIT WORK.

Note: If you delete many rows from a table and then load many new rows into the table in EXPRESS mode, the table grows in size because light appends insert rows at the end of the table, and do not reuse the empty space inside the

table. (Whether or not you specify EXPRESS mode, the loader might choose DELUXE mode to fill in the space if a table has many deleted rows.)

Loading data in DELUXE mode

DELUXE mode combines fast parallel loading with evaluation of indexes and unique constraints. The database server chooses this mode for indexed target tables in all databases, and for STANDARD target tables in logged databases.

About this task

DELUXE mode loads use regular single-row inserts, which add rows to a table that can contain indexes. The insert modifies each index for each row during the load. The insert also checks all constraints for each row. A DELUXE mode load allows you to keep the table unlocked during the load so other users can continue to use it.

You also can use DELUXE mode on tables that do not contain indexes; for instance, if you want to have complete recoverability or maintain access to tables during a load.

You can specify DELUXE mode to override the default EXPRESS load mode for RAW target tables without indexes if the database is logged.

The following steps show you how to prepare a table for DELUXE mode load, create the internal table as type STANDARD, and create the external table.

Tip: The database server will automatically use DELUXE mode to load this type of table. You do not need to specify the DELUXE keyword when you define the external table. If you specify the EXPRESS keyword in the table definition, the database server will use DELUXE mode anyway and issue an informational message.

To use DELUXE-mode load on a table:

1. If you want row locking, specify row locking in the CREATE TABLE statement. (Page locking is the default.) If you want other users to be able to read the table during the load, set the lock mode to `share`. Otherwise, set it to `exclusive`.

   ```
   BEGIN WORK;
   LOCK TABLE employee IN SHARE MODE;
   ```

2. Define the external table.

   ```
   CREATE EXTERNAL TABLE emp_ext
   SAMEAS employee
   USING (
      DATAFILES ("DISK:/work2/mydir/emp.dat"),
      REJECTFILE "/work2/mydir/emp.rej",
      );
   ```

3. Load the table.

   ```
   INSERT INTO employee SELECT * FROM emp_ext;
   ```

4. Commit the load, releasing row or page locks.

```
COMMIT WORK;
```

Results

Important: Configure logical logs to allow maximum concurrent DELUXE load transactions to complete.

Loading from a Delimited File to a Database Table with the Same Schema

You can avoid defining the schema of an external table if it has the same schema as the database table.

About this task

Consider loading a delimited ASCII text file into a table with the following schema:

```
TABLE employee (
   name CHAR(18) NOT NULL,
   hiredate DATE DEFAULT TODAY,
   address VARCHAR(40),
   empno INTEGER);
```

The SQL statements used to load data into the employee table would be as follows:

```
CREATE EXTERNAL TABLE emp_ext
SAMEAS employee
USING (
   DATAFILES ("DISK:/work2/mydir/emp.dat"),
    REJECTFILE "/work2/mydir/emp.rej"
    );
INSERT INTO employee SELECT * FROM emp_ext;
```

The external table has the same name, type, and default for each column because the CREATE statement includes the SAMEAS keyword. The default format is delimited, so no format keyword is required.

Delimited files are ASCII by default. The default row delimiter is an end-of-line character unless you use the RECORDEND keyword to specify a different delimiter when you created the external table. (The RECORDEND keyword works for delimited format only.)

Loading from a Fixed Text File

A fixed text file is one in which data resides in fixed positions within the file.

About this task

The following SQL statements load data from the **emp_exp** external table to a fixed-position table (**employee**):

```
CREATE EXTERNAL TABLE emp_ext
   ( name CHAR(18) EXTERNAL CHAR(18),
     hiredate DATE EXTERNAL CHAR(10),
     address VARCHAR(40) EXTERNAL CHAR(40),
     empno INTEGER EXTERNAL CHAR(6) )
USING (
   FORMAT 'FIXED',
```

```
    DATAFILES ("DISK:/work2/mydir/emp.fix")
    );
INSERT INTO employee SELECT * FROM emp_ext;
```

The enumerated columns use the keyword EXTERNAL to describe the format in which to store the data in the external file.

Loading Between Tables That Have the Same Schema

You can easily move data from an external table to a database table if the tables have the same schema.

About this task

You can load data from one table to another table that has the same schema (for example, **worldemp**) with a simple INSERT statement.

```
INSERT INTO worldemp SELECT * FROM emp_ext;
```

Loading Values into Serial Columns

You can insert successive numbers or explicit values in a serial column.

About this task

The database server loads serial columns with either the values from the original data file or values that the database server automatically generates.

If you want the serial column values to be the values from the data file, the INSERT statement does not require special handling. If you want the database server to generate the value automatically, omit the serial column from the INSERT statement. For example, if the first column in the table (**col1**) is the serial column and you use the following statement, the default mechanism provides the serial value:

```
INSERT INTO mytable (col2, ...) SELECT ...
```

If the table is being loaded into multiple partitions, the serial values are incremented in the same sequence as the table fragments.

Loading Data Warehousing Tables

You can use external tables to load very large tables for data warehousing applications.

This section discusses various scenarios to load very large tables:

- Loading initially
- Refreshing periodically
- Loading of OLTP data from database servers other than Informix®.

Loading Initially

About this task

The following scenario creates and loads a data warehouse table with external data.

To load a table initially

1. Create the table as type RAW to take advantage of light appends and to avoid the overhead of logging during the load.

   ```
   CREATE RAW TABLE tab1 ...
   ```

2. Describe the external data file to the database server with the CREATE EXTERNAL TABLE statement, specifying the EXPRESS statement in the USING clause.
3. Load the table.

   ```
   INSERT INTO tab1 SELECT * FROM ext_tab
   ```

   The table loads quickly, and the operation uses very little log space.

4. Verify the integrity of the data.
5. Create indexes on the table so that queries run more quickly.
6. Perform a level-0 backup so that you can restore the table later, if necessary. You do not need to perform this level-0 backup if it would be just as easy to reload the table from the original source in the case of a problem.

Refreshing Periodically

This scenario loads new data in a data warehouse table periodically from some other source.

About this task

The scenario assumes that the table is type STANDARD during normal operation and that the CREATE EXTERNAL TABLE statement has been previously executed and the EXPRESS keyword was specified in the USING clause.

To refresh a table periodically

1. Drop all indexes on the table.
2. Alter the table to type RAW.

   ```
   ALTER TABLE tab1 TYPE(RAW);
   ```

3. Load the new data in the table.

   ```
   INSERT INTO tab1 SELECT * FROM ext_tab
   ```

   This insert statement quickly appends new data to the end of the table, and the operation uses very little log space.

4. Verify the integrity of the data.
5. Change the table to type STANDARD.

   ```
   ALTER TABLE tab1 TYPE(STANDARD);
   ```

6. Re-create indexes on the table so that queries run more quickly.
7. Perform a level-0 backup to enable you to restore the table later, if necessary. You do not need to perform this level-0 backup if it would be just as easy to reload the table from the original source in the case of a problem.

Initial Loading of OLTP Data from Other Database Servers

This scenario loads data into Informix® for the first time, as you might do when you migrate from a different database server.

About this task

In this scenario, the table to load will be used for OLTP, so you need logged transactions, rollback, and recoverability.

To load OLTP data initially from a different database server using the CREATE EXTERNAL TABLE statement:

1. Create the table as type RAW to take advantage of light appends and to avoid the overhead of logging during the load.

   ```
   CREATE RAW TABLE tab1 ...
   ```

2. Describe the external data file to the database server with the CREATE EXTERNAL TABLE statement.
3. Load the table.

   ```
   INSERT INTO tab1 SELECT * FROM ext_tab
   ```

   The table loads quickly, and the operation uses very little log space.

4. Verify the integrity of the data.
5. Perform a level-0 backup to provide a point from which to recover.
6. Change the table to type STANDARD.

   ```
   ALTER TABLE tab1 TYPE(STANDARD);
   ```

7. Create indexes on the table so that queries run more quickly.
8. Enable constraints on the table to preserve the integrity of the data.

Unloading Data to External Tables from Informix®

You unload data by creating an external table and inserting the data into it, or by selecting data from an internal table into an external file.

To unload data in parallel, initiate a query that runs in parallel and writes its output to multiple files. The unload job uses a round-robin technique to equalize the number of rows in the output files.

Unloading to a Delimited File

About this task

You can unload data to a delimited-ASCII text file from a table, as the following example shows:

```
CREATE EXTERNAL TABLE emp_ext
SAMEAS employee
USING (
    DATAFILES ("DISK:/work2/mydir/emp.dat")
    );
INSERT INTO emp_ext SELECT * FROM employee;
```

Delimited files are ASCII by default.

Unloading to Informix® Data Files

About this task

To unload from the **employee** table to a table in Informix® internal format, use statements similar to the following ones:

```
SELECT * FROM employee
WHERE hiredate > "1/1/1996"
INTO EXTERNAL emp_ext
USING (
   FORMAT 'INFORMIX',
    DATAFILES ("DISK:/work2/mydir/emp.dat")
    );
```

Because the output files use Informix® internal representation, you need to specify the FORMAT 'INFORMIX' option in the USING clause. (The default is delimited-ASCII format.)

Unloading to a Fixed-Text File

You can unload data from the database into fixed format files.

About this task

The following SQL statements unload the **employee** table in fixed text format into the **emp_ext** external table:

```
CREATE EXTERNAL TABLE emp_ext
   ( name CHAR(18) EXTERNAL CHAR(20),
     hiredate DATE EXTERNAL CHAR(10),
     address VARCHAR(40) EXTERNAL CHAR(40),
     empno INTEGER EXTERNAL CHAR(6) )
USING (
   FORMAT 'FIXED',
    DATAFILES ("DISK:/work2/mydir/emp.fix")
    );
INSERT INTO emp_ext SELECT * FROM employee;
```

These statements create a fixed-text file with 20 character positions in the first field, the next 10 character positions in the second field, and so on. Because you are choosing the rows with a SELECT statement, you can format the SELECT list in any way that you want.

Adding an End-of-Line Character to a Fixed Text File

You can add an end-of-line character to each line of a fixed-text file to use the file for other applications.

About this task

If you are writing text in a fixed-text format, separate lines for each record are helpful. An end-of-line character makes the date more legible and clear. If you use delimited format defaults, an end-of-line character is automatic. However, for fixed-format unloads, you need to add an end-of-line character to your records. For example, consider a table with the following schema:

```
TABLE sample (
   lastname CHAR(10),
   firstname CHAR(10),
   dateofbirth DATE);
```

This table contains the following values:

```
Adams      Sam        10-02-1957
```

```
Smith      John       01-01-1920
```

Next, consider an external table with the following schema:

```
CREATE EXTERNAL TABLE sample_ext (
   lastname CHAR(10) EXTERNAL CHAR(10),
   firstname CHAR(10) EXTERNAL CHAR(10),
   dateofbirth DATE EXTERNAL CHAR(12));
```

Unloading **sample_ext** without an end-of-line character produces the following output:

```
Adams     Sam       10-02-1957  Smith     John      01-01-1920
```

You can add end-of-line characters by using a program or script, or by adding a newline field in a SELECT statement.

Using a Program or Script

To add an end-of-line character, you can write the fixed-length records to a data file and then modify the data file with a program or script.

About this task

For example, you could use a C program to find the length of each record, locate the end of a line, and then add an end-of-line character.

Adding a Newline Field in a SELECT Statement

You can use an external table to load the newline character in your internal table.

About this task

To add an end-of-line character, select a final value from a table that contains a newline character, as in the following example:

1. Create a file that contains only a newline character.

   ```
   echo "" > /tmp/cr.fixed
   ```

2. Create an internal table to store this newline value to use when you unload the data.

   ```
   CREATE TABLE dummyCr (cr CHAR(1));
   ```

3. Create the external table to load the newline value.

   ```
   CREATE EXTERNAL TABLE x_cr (cr CHAR(1) EXTERNAL CHAR(1))
   ```

   ```
   USING (DATAFILES ("DISK:/tmp/cr.fixed"), FORMAT 'FIXED');
   ```

4. Load the external table in the internal **dummyCr** table.

```
INSERT INTO dummyCr SELECT * FROM x_cr;
```

Results

The internal table, **dummyCr**, now contains an end-of-line character that you can use to unload in a SELECT statement:

1. To unload data from your internal table to an external table, create the external table with the end-of-line character as an EXTERNAL CHAR.

```
CREATE EXTERNAL TABLE sample_ext
(
   lastname CHAR(10) EXTERNAL CHAR(10),
   firstname CHAR(10) EXTERNAL CHAR(10),
   dateofbirth DATE EXTERNAL CHAR(12),
   eol CHAR(1) EXTERNAL CHAR(1))
USING (DATAFILES ....), FORMAT 'FIXED');
```

2. Select from the internal table and the **dummyCr** table to create an output file that has rows separated by end-of-line characters.

```
INSERT INTO sample_ext(lastname, firstname, dateofbirth, eol)
SELECT a.lastname, a.firstname, a.dateofbirth, b.cr
FROM mytable a, dummyCr b;
```

Restrictions on External Tables

Certain operations on external tables are not supported or have limited scope.

Table 1: Database Tables and External Tables on page 222 compares table operations that are supported for database tables and external tables.

Table 1. Database Tables and External Tables

Table Operation	Database Table	External Table
Support for indexes and: • Primary keys • Foreign keys • Unique and non-unique indexes • Index scans • Automatic index (autoindex)	Yes	No, sequential scans are used.

Table 1. Database Tables and External Tables (continued)

Table Operation	Database Table	External Table
during query execution • Index join		
Triggers are supported	Yes	No
Table can be a target in a MERGE statement	Yes	No. Not allowed as target but allowed as source. See MERGE Example on page 225
Table fragmentation is supported	Yes	No
Multiple database tables are allowed in the FROM clause	Yes	No. See Query Example on page 225
DB-Access LOAD FROM ... INSERT INTO statement is supported	Yes	No
The TRUNCATE TABLE statement truncates a table	Yes	No. Data in external tables is not truncated using the TRUNCATE statement. Unloading data from a database table to an external table automatically truncates the external table.
Table data is replicated	Yes	No
The UPDATE STATISTICS statement is supported	Yes	No
UPDATE and DELETE statements are supported	Yes	No
The ALTER TABLE statement is supported	Yes	No
LBAC is supported	Yes	No

Table 1. Database Tables and External Tables (continued)

Table Operation	Database Table	External Table
Compression is supported	Yes	No
START and STOP VIOLATIONS statements supported	Yes	No
TEMP tables are supported	Yes	No
The EXTERNAL data type is supported for table columns	No	Yes
DEFAULT clause is supported	Yes	No
PUT clause is supported for BLOB and CLOB types	Yes	No. BLOBDIR and CLOBDIR can be specified using the DATAFILES clause.
SERIAL, SERIAL8, and BIGSERIAL data types generate serial numbers	Yes	No. These data types are converted to equivalent integer types and no serial value is generated.
Table can be replicated using Enterprise replication (ER)	Yes	No
Changes to tables are logged and can be replicated	Yes	No. External tables are not logged and cannot be replicated; however system catalogs are replicated.
ACID (atomicity, consistency, isolation, durability) properties are supported	Yes	No
ETL (extract, transform, load) is supported	SQL interface for ETL operations is not supported; however, utilities such as dbload, onload, onunload and LOAD, UNLOAD statements are supported.	Supported using a simple SQL interface using the INSERT ... SELECT statement for high performance loading and unloading of data.

Certain high-availability cluster operations are not supported (see *External Tables in High-Availability Cluster Environments* in the *HCL® Informix® Administrator's Guide*).

To load BLOB or CLOB objects from an external table, you must create a temporary sbspace and create temporary smart large objects in that space to store the BLOB or CLOB data from the external table. Loading BLOB or CLOB data from a read-only secondary server is not supported, because you cannot create a temporary smart large object on a read-only secondary server.

Example

MERGE Example

An external table cannot be the target of the MERGE statement. For example, if **ext** is an external table, the following MERGE statement is valid with **ext** as the source table:

```
MERGE INTO t1
     USING ext ON t1.c1 = ext.c1
    WHEN MATCHED THEN UPDATE
     SET t1.c2 = ext.c2
    WHEN NOT MATCHED THEN INSERT VALUES (99, '999');
```

The following statement, however, fails with **ext** as the target table:

```
  MERGE INTO  ext
     USING t1 ON ext.c1 = t1.c1
    WHEN MATCHED THEN UPDATE
     SET ext.c2 = t1.c2
    WHEN NOT MATCHED THEN INSERT VALUES (99, '999');
```

Example

Query Example

Only the outermost query can have an external table reference. Only one external table can be specified in any query. For example, the following statement is allowed:

```
     SELECT * FROM ext, t2 WHERE ext.c1 = t2.c1;
```

However, the following statements are not allowed:

- Multiple external tables cannot be specified within a query:

  ```
    SELECT * FROM ext, ext3 WHERE ext.c1 = ext3.c1;
  ```

- An external table cannot be used in a subquery:

  ```
     SELECT * FROM t1 WHERE t1.c1  IN (SELECT c1 FROM ext);
  ```

CREATE FUNCTION statement

Use the CREATE FUNCTION statement to create a user-defined function, to register an external function, or to write and register an SPL function.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax



Note:

1. See OR REPLACE Clause on page 229
2. See DBA Keyword and Privileges on the Procedure on page 275
3. Stored Procedure Language only
4. See Routine Parameter List on page 1281
5. See The REFERENCING and FOR Clauses on page 276
6. See Return Clause on page 1271
7. See Specific Name on page 1288
8. See Routine modifier on page 1275
9. See Statement Block on page 1289

10. External routines only
11. See External Routine Reference on page 1229
12. See Quoted String on page 1202

Element	Description	Restrictions	Syntax
function	Name of new function that is defined here	You must have the appropriate language privileges. See GRANT statement on page 537 and Overloading the Name of a Function on page 232.	Identifier on page 1231
owner	Owner of *table_object*	Must own *table_object*	Owner name on page 1262
pathname	Pathname to a file in which compile-time warnings are stored	The specified pathname must exist on the computer where the database resides	The path and filename must conform to your operating-system rules.
table_object	Name or synonym of the table or view whose triggers can call *function*	Must exist in the local database	Identifier on page 1231

Tip: If you are trying to create a function from text of source code that is in a separate file, use the CREATE FUNCTION FROM statement.

Usage

HCL Informix® supports user-defined functions written in these languages:

- The HCL Informix® stored procedure language (SPL).
- One of the external languages (C or Java™) that Informix® supports (*external functions*).

When the IFX_EXTEND_ROLE configuration parameter is set to ON, only users to whom the DBSA grants the built-in EXTEND role can create external functions. Additional requirements for using the CREATE FUNCTION statement are identified in the topic Privileges necessary for using CREATE FUNCTION on page 228.

How many values a function can return is language-dependent. Functions written in SPL can return one or more values. External functions written in the C or Java™ languages must return exactly one value. But a C function can return a collection type, and external functions in queries can return additional values indirectly from OUT parameters (and for the SPL and Java™ languages, from INOUT parameters) that Informix® can process as statement-local variables (SLVs).

Return values from OUT and INOUT parameters of an SPL function can be processed as SLVs. You can also use local variables or parameters of an SPL routine to retrieve values from SPL or C routines that have OUT or INOUT parameters.

For information about how this document uses the terms UDR, function, and procedure, as well as recommended usage, see Relationship Between Routines, Functions, and Procedures on page 273 and Using CREATE PROCEDURE Versus CREATE FUNCTION on page 273.

In ESQL/C, you can use a CREATE FUNCTION statement only within a PREPARE statement. If you want to create a user-defined function for which the text is known at compile time, you must put the text in a file and specify this file with the CREATE FUNCTION FROM statement.

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if a function of the specified name is already registered in the current database. (Because the identifier of a function can be overloaded, it might be unnecessary to include these keywords, if the database server can resolve the argument list of the new function as different from that of any other function of the same name in the current database.)

Functions use the collating order that was in effect when they were created. See SET COLLATION statement on page 775 for information about using non default collation.

Privileges necessary for using CREATE FUNCTION

You must hold the Resource privilege or the DBA privilege on a database to create a function within that database.

Before you can create a function, you must also hold the Usage privilege on the programming language in which the function is written. The GRANT USAGE ON LANGUAGE statement can specify the SPL, C, or Java™ language when it grants a language-level privilege to a user or to a role. For more information, see Language-Level Privileges on page 549.

By default, Usage privilege on SPL is granted to PUBLIC.

To register functions in the C or Java™ external programming languages, you must also hold the built-in EXTEND role, unless the IFX_EXTEND_ROLE configuration parameter is set to `0` or to `Off`.

DBA keyword and Execute privilege on the created function

If you create a UDR with the DBA keyword, it is known as a DBA-privileged UDR. You need the DBA privilege to create a DBA-privileged UDR. If you omit the DBA keyword, the UDR is known as an owner-privileged UDR. Other users who have Execute privilege on the UDR run the UDR with the privileges of the user who created the UDR.

DBA-privileged UDRs

For a DBA-privileged UDR, users who do not hold the DBA privilege can run the routine under the following conditions:

- The DBA grants the user Execute privilege on the UDR.
- The DBA grants the Execute privilege to PUBLIC so that all users can run the UDR.

When you have Execute privilege on a DBA-privileged UDR, you run the UDR with DBA privileges.

For additional information about DBA-privileged UDRs, see Ownership of Created Database Objects on page 281.

Owner-privileged UDRs

In an ANSI-compliant database, before other users can execute an owner-privileged UDR, the owner must grant the Execute privilege, either to individual users, or to roles, or to PUBLIC.

In a database that is not ANSI compliant, any user can run the UDR because PUBLIC is granted the Execute privilege by default. To restrict access to an owner-privileged UDR to specific users, the owner must revoke the Execute privilege on the UDR from PUBLIC, and then grant it to specified users or roles. However, you can prevent privileges on UDRs from being granted to PUBLIC by default. by setting the **NODEFDAC** environment variable to `yes`. Then the owner must grant the Execute privilege for the UDR to other users.

For an owner-privileged UDR, all non-qualified database objects that the UDR references are implicitly qualified by the name of the UDR owner. To allow other users to run the UDR, the owner of the UDR must be granted the necessary privileges on any underlying database objects with the WITH GRANT OPTION keyword in the GRANT statement.

The privileges of the owner determine who can run an owner-privileged UDR:

- If the owner does not have privileges on all of the objects that are referenced by the UDR, then the UDR fails regardless of whether the owner or another user runs it.
- If the owner has privileges on all referenced objects, but does not have the GRANT option, then only the owner can run the UDR. Other users cannot run the UDR, even if they have Execute privilege on the UDR.
- If the owner has privileges with the GRANT option on all referenced objects, then the owner and other users who have the Execute privilege can run the UDR. Other users who have Execute privilege run the UDR with the owner's privileges.

If an external C or Java™ language function has a negator function, you must grant the Execute privilege on both the external function and on its negator function before users can execute the external function.

OR REPLACE Clause

The OR REPLACE clause is used to update an existing function without changing any object privileges granted on it.

Example

```
create function testfunc() returning char(50);
    return "this is the original function";
end function;
Routine created.

execute function testfunc();

(expression)

this is the original function

1 row(s) retrieved.

create function testfunc() returning char(50);
    return "this is the replaced function";
```

```
end function;

  673: Another routine (testfunc) with same signature already exists in database


create or replace function testfunc() returning char(50);
    return "this is the replaced function";
end function;
Routine created.

execute function testfunc();

(expression)

this is the replaced function

1 row(s) retrieved.
```

The REFERENCING and FOR Clauses

The REFERENCING clause can declare correlation names for the original value and for the updated value in columns of the *table_object* that the FOR clause specifies.

(explicit id unique_321_Connect_42_r4crf002) unique_321_Connect_42_r4crf002

REFERENCING and FOR Clauses

" **REFERENCING** "

" { **OLD** | **NEW** } [**AS**] *correlation* "

" **FOR** [*'owner'* .] *table_object* "

Element	Description	Restrictions	Syntax
correlation	Name that you declare here to qualify an old or new column value (as *correlation*.*column*) in a trigger routine	Must not be *table_object*	Identifier on page 1231
owner	Owner of *table_object*	Must own *table_object*	Owner name on page 1262
table_object	Name or synonym of a table or view whose triggers can call *function*	Must exist in the local database	Identifier on page 1231

If you include the REFERENCING and FOR *table_object* clauses immediately after the parameter list of the CREATE FUNCTION statement, the function that you create is known as a *trigger function* (or *trigger UDR* or *trigger routine*). The FOR

clause specifies the table or view whose triggers can invoke the function from the FOR EACH ROW section of their Triggered Action list.

In the REFERENCING clause, the OLD *correlation* specifies a prefix by which the trigger routine can reference the value that a column of *table_object* had before the trigger routine modifies that column value. The NEW *correlation* specifies a prefix for referencing the new value that the trigger routine assigns to the column. Whether the trigger routine can use correlation names to reference the OLD column value, the NEW column value, or both values depends on the type of triggering event:

- A trigger routine invoked by an Insert trigger can reference only the NEW correlation name.
- A trigger routine invoked by a Delete trigger or by a Select trigger can reference only the OLD correlation name.
- A trigger routine invoked by an Update trigger can reference both the OLD and the NEW correlation names.

For information about how to use the *correlation.column* notation in triggered actions, see REFERENCING Clauses on page 395.

Besides the general requirements for any Informix® UDR that is written in the SPL language, trigger routines can support certain additional syntax features, and are subject to certain restrictions, that are not features (or that are not restrictions) for ordinary UDRs that are not trigger routines:

- A trigger routine must include the FOR *table_object* clause that specifies the name of the table or view in the local database whose triggers can invoke this routine.
- A trigger routine can also include the REFERENCING clause to declare correlation names for OLD and NEW values that SPL statements in the UDR can reference.
- Trigger routines can be invoked only in the FOR EACH ROW section of the Triggered Action list in the trigger definition.
- Correlated variables for OLD or NEW values can appear in the IF statement of SPL and in CASE expressions.
- Correlated variables for OLD values cannot be on the left-hand side of a LET expression.
- Correlated variables for NEW values cannot be on the left-hand side of a LET expression if the FOR clause specifies a view whose INSTEAD OF trigger action list invokes the trigger routine.
- Only correlated variables for NEW values can be on the left-hand side of a LET expression that references correlated variables. In this case, however, the FOR clause must specify a table, rather than a view.
- Both OLD and NEW values can be on the right-hand side of a LET expression.
- The Boolean operators SELECTING, INSERTING, DELETING, and UPDATING are valid in trigger routines (and only in trigger routines and in other UDRs that are invoked in triggered action statements) in contexts where Boolean expressions are valid. These operators return TRUE ('t') if the triggering event matches the DML operation referenced by the name of the operator, and they return FALSE ('f') otherwise.
- If a single triggering event activates multiple triggers on the same table or view, then all of the BEFORE actions take place before any of the FOR EACH ROW actions, and all of the AFTER actions follow the FOR EACH ROW actions. The order of execution of different triggers on the same event is not guaranteed.

- Trigger routines must be written in the SPL language. They cannot be written in an external language, like the C or Java™ language, but they can include calls to external language routines, such as the **mi_trigger** application programming interface for trigger introspection.
- Trigger functions cannot reference savepoints. Any changes to the data values or to the schema of the database by a triggered action must be committed or rolled back in their entirety. Informix® does not support the partial rollback of triggered actions.

For more information about the **mi_trigger** API, refer to the *HCL® Informix® DataBlade® API Programmer's Guide* and to the *IBM® Informix® DataBlade® API Function Reference*.

If you include the REFERENCING clause but omit the FOR clause, or if you include the FOR clause but omit the REFERENCING clause, the CREATE FUNCTION statement fails with an error.

If you omit the REFERENCING and FOR clauses, the UDR cannot use the SELECTING, INSERTING, DELETING, and UPDATING operators, and cannot declare variables that can represent and manipulate column values in triggered actions on the table or view that the trigger definition specifies.

See the REFERENCING Clauses on page 395 section in the CREATE TRIGGER statement description for the syntax of the REFERENCING clause for Delete, Insert, Select, and Update triggers.

Overloading the Name of a Function

Because Informix® supports *routine overloading*, you can define more than one function with the same name, but different parameter lists. You might want to overload functions in the following situations:

- You create a user-defined function with the same name as a built-in function (such as **equal( )**) to process a new user-defined data type.
- You create *type hierarchies*, in which subtypes inherit data representation and functions from supertypes.
- You create *distinct types*, which are data types that have the same internal storage representation as an existing data type, but have different names and cannot be compared to the source type without casting. Distinct types inherit support functions from their source types.

For a brief description of the routine signature that uniquely identifies each user-defined function, see Routine Overloading and Routine Signatures on page 1228.

Examples

Overloaded functions are uniquely identified by the name and the input parameter list. Instead of providing a long unique identifier, it is possible to provide specific name and use it later. The following example illustrates an overloaded function, whose identifier is**getArea**, that has the specific names **getSquareArea** and **getRectangleArea**:

```
CREATE FUNCTION getArea
  (i INT DEFAULT 0)
RETURNING INT SPECIFIC getSquareArea;
DEFINE j INT;
LET j = i * i;
RETURN j;
```

```
END FUNCTION;

CREATE FUNCTION getArea
  (i INT DEFAULT 0, j INT DEFAULT 0)
RETURNING INT SPECIFIC getRectangleArea;
 DEFINE k INT;
 LET k = i * j;
 RETURN k;
END FUNCTION;
```

Now you can use the specific name, as in the following example:

```
GRANT EXECUTE ON SPECIFIC FUNCTION getSquareArea TO informix;
GRANT EXECUTE ON SPECIFIC FUNCTION getRectangleArea TO informix;
```

Without the specific name, you would need to issue the following:

```
GRANT  EXECUTE ON FUNCTION getArea (INTEGER) TO informix;
GRANT  EXECUTE ON FUNCTION getArea (INTEGER,INTEGER) TO informix;
```

Using the SPECIFIC Clause to Specify a Specific Name

You can declare a specific name, unique to the database, for a user-defined function. A specific name is useful when you are overloading a function.

DOCUMENT Clause

The quoted string in the DOCUMENT clause provides a synopsis and description of the UDR. The string is stored in the **sysprocbody** system catalog table and is intended for the user of the UDR. Anyone with access to the database can query the **sysprocbody** system catalog table to obtain a description of one or all of the UDRs stored in the database.

For example, the following query obtains a description of the SPL function **update_by_pct**, that SPL Functions on page 234 shows:

```
SELECT data FROM sysprocbody b, sysprocedures p
WHERE b.procid = p.procid
      --join between the two catalog tables
   AND p.procname = 'update_by_pct'
      -- look for procedure named update_by_pct
   AND b.datakey  = 'D'-- want user document;
```

The preceding query returns the following text:

```
USAGE: Update a price by a percentage
Enter an integer percentage from 1 - 100
and a part id number
```

A UDR or application program can query the system catalog tables to fetch the DOCUMENT clause and display it for a user.

For C and Java™ language functions, you can include a DOCUMENT clause at the end of the CREATE FUNCTION statement, whether or not you use the END FUNCTION keywords.

SPL Functions

SPL functions are UDRs written in SPL that return one or more values. To write and register an SPL function, use a CREATE FUNCTION statement. Embed appropriate SQL and SPL statements between the CREATE FUNCTION and END FUNCTION keywords. You can also follow the function with the DOCUMENT and WITH FILE IN options.

SPL functions are parsed, optimized (as far as possible), and stored in the system catalog tables in executable format. The body of an SPL function is stored in the **sysprocbody** system catalog table. Other information about the function is stored in other system catalog tables, including **sysprocedures**, **sysprocplan**, and **sysprocauth**. For more information about these system catalog tables, see the *HCL® Informix® Guide to SQL: Reference*.

The END FUNCTION keywords are required in every SPL function, and a semicolon (**;**) must follow the clause that immediately precedes the statement block. The following code example creates an SPL function:

```
CREATE FUNCTION update_by_pct ( pct INT, pid CHAR(10))
   RETURNING INT;
   UPDATE inventory SET price = price + price * (pct/100)
      WHERE part_id = pid;
   return (select price from inventory where part_id = pid);
END FUNCTION
   DOCUMENT "USAGE: Update a price by a percentage",
         "Enter an integer percentage from 1 - 100",
         "and a part id number"
   WITH LISTING IN '/tmp/warn_file';
```

For more information on how to write SPL functions, see the chapter about SPL routines in *HCL® Informix® Guide to SQL: Tutorial***.**

See also the section Transactions in SPL Routines on page 1294.

You can include valid SQL or SPL language statements in SPL functions. See, however, the following sections in Other syntax segments on page 1209 that describe restrictions on SQL and SPL statements within SPL routines: Subset of SPL Statements Valid in the Statement Block on page 1290; SQL Statements Valid in SPL Statement Blocks on page 1291; and Restrictions on SPL Routines in Data-Manipulation Statements on page 1293.

External Functions

About this task

External functions are functions you write in an external language (that is, a programming language other than SPL) that Informix® supports.

To create a C user-defined function

1. Write the C function.
2. Compile the function and store the compiled code in a shared library (the shared-object file for C).
3. Register the function in the database server with the CREATE FUNCTION statement.

Results

To create a user-defined function written in the Java™ language

1. Write a Java™ static method, which can use the JDBC functions to interact with the database server.
2. Compile the Java™ source file and create a **.jar** file (the shared-object file for Java™).
3. Execute the **install_jar( )** procedure with the EXECUTE PROCEDURE statement to install the JAR file in the current database.
4. If the UDR uses user-defined data types, create a map between SQL data types and Java™ classes. Use the **setUDTextName( )** procedure that is explained in EXECUTE PROCEDURE statement on page 510.
5. Register the UDR with the CREATE FUNCTION statement.

Rather than storing the body of an external routine directly in the database, the database server stores only the pathname of the shared-object file that contains the compiled version of the routine. When it executes the external routine, the database server invokes the external object code.

The database server stores information about an external function in system catalog tables, including **sysprocbody** and **sysprocauth**. For more information on the system catalog, see the *HCL® Informix® Guide to SQL: Reference*.

Example of Registering a C User-Defined Function

The following example registers an external C user-defined function named **equal( )** in the database. This function takes two arguments of the type **basetype1** and returns a single Boolean value. The external routine reference name specifies the path to the C shared library where the function object code is actually stored. This library contains a C function **basetype1_equal( )**, which is invoked during execution of the **equal( )** function.

```
CREATE FUNCTION equal ( arg1 basetype1, arg2 basetype1)
   RETURNING BOOLEAN;
   EXTERNAL NAME
      "/usr/lib/basetype1/lib/libbtype1.so(basetype1_equal)"
   LANGUAGE C
END FUNCTION;
```

Example of Registering a UDR Written in the Java™ Language

The following CREATE FUNCTION statement registers the user-defined function, **sql_explosive_reaction( )**. This function is discussed in sqlj.install_jar.

```
CREATE FUNCTION sql_explosive_reaction(INT) RETURNS INT WITH (class="jvp")
   EXTERNAL NAME "course_jar:Chemistry.explosiveReaction" LANGUAGE JAVA;
```

This function returns a single INTEGER value. The EXTERNAL NAME clause specifies that the Java™ implementation of the **sql_explosive_reaction( )** function is a method called **explosiveReaction( )**, which resides in the **Chemistry** Java™ class that resides in the **course_jar** JAR file.

Ownership of Created Database Objects

The user who creates an owner-privileged UDR, rather than the user who executes the UDR, owns any database objects that are created by the UDR when the UDR is executed, unless another owner is specified for the created object.

For example, assume that user **mike** creates this user-defined function:

```
CREATE FUNCTION func1 () RETURNING INT;
   CREATE TABLE tab1 (colx INT);
   RETURN 1;
END FUNCTION;
```

If user **joan** now executes function **func1**, user **mike**, not user **joan**, is the owner of the newly created table **tab1**.

In the case of a DBA-privileged UDR, however, the user who executes a UDR (rather than the UDR owner) owns any database objects created by the UDR, unless another owner is specified for the database object within the UDR.

For example, assume that user **mike** creates this user-defined function:

```
CREATE DBA FUNCTION func2 () RETURNING INT;
   CREATE TABLE tab2 (coly INT);
   RETURN 1;
END FUNCTION;
```

If user **joan** now executes function **func2**, user **joan**, not user **mike**, is the owner of the newly created table **tab2**.

See also the section Support for roles and user identity on page 1294.

CREATE FUNCTION FROM statement

Use the CREATE FUNCTION FROM statement to access a user-defined function whose CREATE FUNCTION statement resides in a separate file.

This statement is an extension to the ANSI/ISO standard for SQL. Use this statement with ESQL/C.

Syntax

```
CREATE FUNCTION FROM [IF NOT EXISTS] { 'file' | file_var }
```

Element	Description	Restrictions	Syntax
file	Path and filename of a file that contains the full CREATE FUNCTION statement text. Default pathname is current directory.	Must exist and contain exactly one CREATE FUNCTION statement	Must conform to operating-system rules.
file_var	Variable storing value of *file*	Same as for *file*	Language specific

Usage

Functions written in the C or Java™ language are called *external* functions. When the IFX_EXTEND_ROLE configuration parameter is set to ON, only users who have been granted the built-in EXTEND role can create external functions.

programs cannot directly create user-defined functions. That is, they cannot contain the CREATE FUNCTION statement.

To create these functions within programs:

1. Create a source file with the CREATE FUNCTION statement.
2. Use the CREATE FUNCTION FROM statement to send the contents of this source file to the database server for execution.

   The file that you specify in the *file* parameter can contain only one CREATE FUNCTION statement.

For example, suppose that the following CREATE FUNCTION statement is in a separate file, called **del_ord.sql**:

```
CREATE FUNCTION delete_order( p_order_num INT) RETURNING INT, INT;
   DEFINE item_count INT;
   SELECT count(*) INTO item_count FROM items
      WHERE order_num = p_order_num;
   DELETE FROM orders WHERE order_num = p_order_num;
   RETURN p_order_num, item_count;
END FUNCTION;
```

In the program, you can access the **delete_order( )** SPL function with the following CREATE FUNCTION FROM statement:

```
EXEC SQL create function from 'del_ord.sql';
```

If you are not sure whether the UDR in the file is a user-defined function or a user-defined procedure, use the CREATE ROUTINE FROM statement.

The filename that you provide is relative. If you provide a simple filename with no pathname (as in the preceding example), the client application looks for the file in the current directory.

Important: The preprocessor does not process the contents of the file that you specify. It only sends the contents to the database server for execution. Therefore, there is no syntactic check that the file that you specify in CREATE FUNCTION FROM actually contains a CREATE FUNCTION statement. To improve readability of the code, however, it is recommended that you match these two statements.

CREATE INDEX statement

Use the CREATE INDEX statement to create an index for one or more columns in a table, or on values returned by a UDR that uses column values as arguments.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
(explicit id unique_339_Connect_42_crein001) unique_339_Connect_42_crein001(explicit id unique_339_Connect_42_crein002)
unique_339_Connect_42_crein002(explicit id unique_339_Connect_42_crein003) unique_339_Connect_42_crein003(explicit id
unique_339_Connect_42_crein004) unique_339_Connect_42_crein004(explicit id unique_339_Connect_42_crein005) unique_339_Connect_42_crein005(explicit
id unique_339_Connect_42_crein006) unique_339_Connect_42_crein006(explicit id unique_339_Connect_42_crein013)
unique_339_Connect_42_crein013(explicit id unique_339_Connect_42_crein012) unique_339_Connect_42_crein012(explicit id
unique_339_Connect_42_crein011) unique_339_Connect_42_crein011(explicit id unique_339_Connect_42_crein015) unique_339_Connect_42_crein015(explicit
id unique_339_Connect_42_crein016) unique_339_Connect_42_crein016 CREATE <Index-Type Options> [] INDEX [IF NOT EXISTS] index ON {
table | synonym } <Index-Key Specs> [] <Index Options>
```

Index Options

- " [<USING Access-Method Clause> []] "
- " [<FILLFACTOR Option> []] [<Storage Options> []] "
- " [<Index Modes> []] [HASH ON Clause []] "
- " [ONLINE []] "
- " [COMPRESSED []] "
- " [<Extent Size Options> []] "
- " [Index visibility options []] "

Element	Description	Restrictions	Syntax
index	The name declared here for a new index.	The name must be unique among names of indexes in the database.	Identifier on page 1231
synonym, *table*	The name or synonym of a standard or temporary *table* to be indexed	The synonym and its table must exist in the current database. **Note:** BTS index on temp table is not supported.	Identifier on page 1231

Usage

When you issue the CREATE INDEX statement, the table is locked in exclusive mode. If another process is using the table, CREATE INDEX returns an error. (For an exception, however, see The ONLINE keyword of CREATE INDEX on page 260.)

If the index is on a column that stores encrypted data, the database server cannot use the index.

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if an index of the specified name is already defined on the specified table in the current database.

Indexes use the collation that was in effect when CREATE INDEX executed.

A *secondary-access method* (sometimes referred to as an *index-access method*) is a set of database server functions that build, access, and manipulate an index structure such as a B-tree, R-tree, or an index structure that a module provides, in order to speed up the retrieval of data.

Neither *synonym* nor *table* can refer to a virtual table or to a table object that the CREATE EXTERNAL TABLE statement defined.

You cannot directly base a functional index on a built-in function, but you can create an SPL wrapper that calls and returns a value from a built-in function. The arguments to a user-defined function that defines a functional index cannot be the values from a column of a collection data type.

The following statistics are generated automatically by the CREATE INDEX statement, with or without the ONLINE keyword:

- Index-level statistics, equivalent to the statistics gathered in the UPDATE STATISTICS operation in LOW mode for B-tree indexes.
- Column-distribution statistics, equivalent to the distribution generated in the UPDATE STATISTICS operation in HIGH mode, for a non-opaque leading indexed column of an ordinary B-tree index.

Index-type options

Use the DISTINCT or UNIQUE and CLUSTER options of the CREATE INDEX statement to specify the characteristics of the index.

Index-Type Options

"[{ DISTINCT | UNIQUE }]"

"[CLUSTER]"

DISTINCT

Specifies that the columns on which the index is based accept only unique data.

UNIQUE

Specifies that the columns on which the index is based accept only unique data.

CLUSTER

Reorders the rows of the table in the order that the index designates.

UNIQUE or DISTINCT option usage

If you do not specify the UNIQUE or DISTINCT keyword, the index allows duplicate values in the indexed column or in the set of indexed columns.

A column with a unique index can have, at most, one NULL value.

You cannot specify an R-tree secondary-access method for a UNIQUE index key.

The following example creates a unique index that prevents duplicate values in the **customer_num** column:

```
CREATE UNIQUE INDEX c_num_ix ON customer (customer_num);
```

The DISTINCT and UNIQUE keywords are synonyms, so the following statement has the same effect as the previous example:

```
CREATE DISTINCT INDEX c_num_ix ON customer (customer_num);
```

The index in both examples is maintained in ascending order, which is the default order. The next example defines a unique descending index called **c_num_desc_ix** on the same column:

```
CREATE UNIQUE INDEX c_num_desc_ix ON customer (customer_num DESC);
```

You can also prevent duplicate values in a column or in a set of columns by creating a unique constraint with the CREATE TABLE or ALTER TABLE statement and the ADD CONSTRAINT clause.

In an NLSCASE INSENSITIVE database, indexes on columns of the NCHAR and NVARCHAR data types disregard lettercase differences, so that the database server treats case variants among strings composed of the same sequence of letters as duplicate values. You cannot insert or update a row of table with an NCHAR or NVARCHAR column on which a unique index or a unique constraint is defined, if that column value in the new row differs only by letter case from the value in the same column of any existing row of the same table. For more information about databases with the NLSCASE INSENSITIVE property, see Duplicate rows in NLSCASE INSENSITIVE databases on page 699 and NCHAR and NVARCHAR expressions in case-insensitive databases.

CLUSTER option usage

You cannot specify the CLUSTER option and the ONLINE keyword in the same statement. In addition, some secondary-access methods (such as R-tree) do not support clustering. Before you specify CLUSTER for your index, be sure that the index uses an access method that supports clustering.

The CREATE CLUSTER INDEX statement fails if a CLUSTER index already exists on the same table.

```
CREATE CLUSTER INDEX c_clust_ix ON customer (zipcode);
```

This statement creates an index on the **customer** table and physically orders the rows according to their postal code values, in (by default) ascending order.

If the CLUSTER option is specified and fragments exist on the data, values are clustered only within each fragment, and not globally across the entire table.

If the CREATE CLUSTER INDEX statement also includes the COMPRESSED keyword as a storage option, the database server issues error -26950. To create a cluster index that supports compression requires two steps:

- Use the CREATE CLUSTER INDEX statement to define a cluster index with no index compression.
- Call the SQL administration API **task( )** or **admin( )** function with the `'index compress'` argument to compress the existing cluster index.

You cannot use the CLUSTER option on a forest of trees index.

How indexes affect primary-key, unique, and referential constraints

The database server creates internal B-tree indexes for primary-key, unique, and referential constraints. If a primary-key, unique, or referential constraint is added after the table is created, any user-created indexes on the constrained columns are used, if appropriate.

An appropriate index is one that indexes the same columns that are used in the primary-key, referential, or unique constraint.

If an appropriate user-created index is not available, the database server creates a nonfragmented internal index on the constrained column or columns.

Index-key specification

Use the Index-key specification of the CREATE INDEX statement to define the key value for the index. This can also specify the ascending or descending sort order, and the operator class.

This is the syntax of the Index-Key Specification:

(explicit id unique_341_Connect_42_bson1) unique_341_Connect_42_bson1

Index-Key Specification

" ({ { *column* | *function* (*func_col*) } | [*op_class*] | <BSON column specification> [] } { **ASC** | **[DESC]** }) "

Element	Description	Restrictions	Syntax
column	Column whose value is used as a key to this index	See Restrictions on columns as index keys on page 244.	Identifier on page 1231
function	User-defined function whose return value is used as a key to this index	Must be a nonvariant function that does not return a large object data type. Cannot be a built-in algebraic, exponential, log, or hex function.	Identifier on page 1231
func_col	Column whose value is an argument to *function*	Cannot be of a collection data type. See Using the return value of a function as an index key on page 245.	Identifier on page 1231

Element	Description	Restrictions	Syntax
op_class	Operator class associated with *column* or *function* for this index key	If the secondary-access method in the USING clause has no default operator class, you must specify one here. (See Using an Operator Class on page 249.)	Identifier on page 1231

The index-key value can be one or more columns of built-in data types. If you specify multiple columns, the concatenation of values from the set of columns is treated as a single composite column for indexing.

The index-key value also can be one of the following:

- A column of type LVARCHAR(*size*), if *size* is smaller than 380 bytes
- One or more columns of user-defined data types
- One or more values that a user-defined function returns (referred to as a *functional index*), where the argument list of the UDF is one or more column values in the same row
- A combination of one or more column values and the return value from one or more user-defined functions.

The 380-byte LVARCHAR size limit is for dbspaces of the default (2 kilobyte) page size, but dbspaces of larger page sizes can support larger index key sizes, as listed in the following table.

Table 2. Maximum Index Key Size for Selected Page Sizes

Page Size	Maximum Index Key Size
2 kilobytes	380 bytes
8 kilobytes	1,609 bytes
16 kilobytes	3,247 bytes
32 kilobytes	6,520
64 kilobytes	13,074
128 kilobytes	26,181
256 kilobytes	Max row size (32 kilobytes)

Specifying the sort order

By default, the index is sorted in ascending order, from the lowest value to the highest, according to the collation order for the locale, or else to the collation order that was in effect when the index was created, if the SET COLLATION statement has specified a nondefault collation. You can use the DESC keyword to reverse the sort order, so that the index is sorted from the highest value to the lowest.

If you explicitly specify the ASC keyword in the Index-Key Specification, the index is sorted in ascending order.

Specifying an operator class

If the secondary access method in the USING clause has no default operator class, the Index-Key Specification can specify an operator class for the index key.

If the secondary access method in the USING clause has a default operator class, the Index-Key Specification can specify an operator class to override the default operator class for the index.

Indexing a BSON field

You can create an index on a field within a BSON column.

Syntax

Use the following syntax in the index-key specification. See Index-key specification on page 241.

BSON field specification

```
" { BSON_GET | BSON_value_function } ( " BSON_column_name . { BSON_field_name | BSON_array_position }
" ) "

" USING BSON "
```

Element	Description	Restrictions	Syntax
BSON_array_position	A positive integer that represents the position of a value in an array, starting with 0 for the first value.	Must be preceded by all ancestor field names.	
BSON_column_name	A BSON column name	Must be of type BSON.	Expression on page 1003
BSON_field_name	BSON field name	Must be a literal BSON field name. Can be a multilevel field identifier, up to 32 levels. All ancestor field names must be included.	Column Expressions on page 1025
BSON_value_function	A BSON value function for a specific data type, except the BSON_VALUE_OBJECTID function	You cannot use the BSON_VALUE_OBJECTID function to create an index	BSON processing functions

You cannot create an index on a BSON column. You must create the index on a field within the BSON column.

The BSON_GET or BSON value function specifies which field to index. The USING BSON keywords are necessary to specify that the index is created on a BSON column.

Example

Example: Create an index on a BSON field

The following statements create and populate a table that has a BSON column:

```
CREATE DATABASE testdb WITH LOG;
CREATE TABLE IF NOT EXISTS bson_table(bson_col BSON);

INSERT INTO bson_table VALUES(
    '{person:{givenname:"Jim",surname:"Flynn",age:29,cars:["dodge","olds"]}}'
    ::JSON::BSON);
```

The following statement creates an index on the **surname** field in the BSON column:

```
CREATE INDEX idx2 ON bson_table(
   BSON_GET(bson_col, "person.surname")) USING BSON;
```

Restrictions on columns as index keys

Whether the database server accepts the Index Key specification as valid can depend on various factors, including attributes of the table, and the data types, the storage size, and the total number of index-key columns, as well as dependencies between index-key columns and constraints on the same table.

The following restrictions apply to any column or column list that the Index Key Specification of the CREATE INDEX statement references:

- All the columns must exist in the table on which the index is defined.
- The table must exist in the current database, and cannot be an object that the CREATE EXTERNAL TABLE statement defined.
- A column defined as an index key cannot be a complex data type (LIST, MULTISET, SET, a generic COLLECTION type, or a ROW type) or a JSON or BSON data type, with the following exceptions:
    - You can create a *functional index* on a column of a named ROW type.
    - You can create an index on a specific field of a BSON column, or create multiple indexes, each on a different field of the same BSON column. You cannot create an index on an entire BSON column.
- The maximum number of columns and the total width of all column index keys are dependent on the page size of the database server. See Creating Composite Indexes on page 245.
- You cannot add an ascending index to a column list on which a unique constraint is defined. See Using the ASC and DESC Sort-Order Options on page 246.
- You cannot add a unique index to a column list that has a primary-key constraint. The reason is that defining the column or column list as the primary key causes the database server to implement the constraint by creating a unique internal index on the column or column list. The CREATE INDEX statement cannot define another unique index whose key is the same column or column list.
- The number of indexes that you can define on the same list of columns is restricted. See Restrictions on the Number of Indexes on a Set of Columns on page 248.

For additional index-key restrictions that apply to columns that are specified as arguments to functional indexes, see Using the return value of a function as an index key on page 245.

Using the return value of a function as an index key

A *functional index* is indexed on the value that a specified function returns from a column-value argument, rather than on the value of a column.

For example, the following statement creates a functional index on table **zones** using the value that the user-defined function **Area( )** returns as the key:

```
CREATE INDEX zone_func_ind ON zones (Area(length,width));
```

You can create the function on which to define a functional index within an SPL routine. You can also create an index on a nonvariant user-defined function that does not return a large object.

The functional index can be a B-tree index, an R-tree index, or a user-defined secondary-access method.

The value returned by the function can be the index key, as in the example above, or it can be part of a composite index whose other key parts are the values of columns, the values of partial columns, or the return values of other functional indexes. (For more information, see the topic Creating Composite Indexes on page 245).

Important: The database server imposes the following restrictions on the user-defined routines (UDRs) on which a functional index is defined:

- The arguments cannot be the names of column of a collection data type (LIST, MULTISET, or SET).
- The function cannot return a large object of the data types BLOB, BYTE, CLOB, and TEXT.
- The function cannot be a VARIANT function.
- The function cannot include any DML statement of SQL.
- The ONLINE keyword is not valid when you create a functional index; see The ONLINE keyword of CREATE INDEX on page 260.
- The function must be a user-defined function, or, to create an index on a field in a BSON column, the BSON_GET function or a BSON value function. You cannot create a functional index on any other built-in function of SQL.

Despite the last restriction above, however, you can create a functional index on a user-defined function that calls a non-variant built-in SQL function, so that the value returned by the built-in function is the index key of a functional index. (That is, create an SPL wrapper that calls and returns the value from a built-in function of SQL, and then define a functional index on this user-defined SPL function.)

Creating Composite Indexes

A *simple* index lists only one *column* (or only one *function*, whose argument list must be a list of one or more columns) in its Index Key Specification. Any other index is a *composite* index. You should list the columns in a composite index in the order from most frequently used to least frequently used.

If you use SET COLLATION to specify the collating order of a nondefault locale, you can create multiple indexes on the same set of columns, using different collations. (Such indexes are useful only on NCHAR or NVARCHAR columns.)

The following example creates a composite index using the **stock_num** and **manu_code** columns of the **stock** table:

```
CREATE UNIQUE INDEX st_man_ix ON stock (stock_num, manu_code);
```

The UNIQUE keyword prevents any duplicates of a given combination of **stock_num** and **manu_code**. The index is in ascending order by default.

You can include up to 16 columns in a composite index. The total width of all indexed columns in a single composite index cannot exceed 380 bytes.

An *index key part* is either a column in a table, or the result of a user-defined function on one or more columns. A composite index can have up to 16 key parts that are columns, or up to 341 key parts that are values returned by a UDR. This limit is language-dependent and applies to UDRs written in SPL or Java™. Functional indexes based on C language UDRs can have up to 102 key parts. A composite index can include any of the following index key parts in its index key:

- One or more columns
- One or more values that a user-defined function returns (referred to as a *functional index*).

The index key parts of a composite index can be a combination of columns and user-defined functions.

For dbspaces of the default page size of 2 kilobytes, the total width of all indexed columns in a single CREATE INDEX statement cannot exceed 380 bytes, except for functional indexes of Informix®, whose language-dependent limits are described earlier in this section. For the maximum sizes in dbspaces larger than 2 kilobytes, see Index-key specification on page 241.

Whether the index is based directly on column values in the table, or on functions that take column values as arguments, the maximum size of the index key depends only on page size. The maximum index key size for functional indexes in dbspaces larger than 2 kilobytes are the same as for column indexes. The only difference between limits on column indexes and functional indexes is the number of key parts. An index based on columns can have no more than 16 key parts, but a functional index has different language-dependent limits on key parts. For a given page size, the maximum index key size is the same for both column-based and functional indexes.

Using the ASC and DESC Sort-Order Options

The ASC option specifies an index maintained in ascending order; this is the default order. The DESC option can specify an index that is maintained in descending order. These ASC and DESC options are valid with B-trees only.

Effects of Unique Constraints on Sort Order Options

When a column or list of columns is defined as unique in a CREATE TABLE or ALTER TABLE statement, the database server implements that UNIQUE CONSTRAINT by creating a unique ascending index. Thus, you cannot use the CREATE INDEX statement to add an ascending index to a column or column list that is already defined as unique.

However, you can create a descending index on such columns, and you can include such columns in composite ascending indexes in different combinations. For example, the following sequence of statements is valid:

```
CREATE TABLE customer (
   customer_num  SERIAL(101) UNIQUE,
   fname                CHAR(15),
   lname                CHAR(15),
   company              CHAR(20),
   address1             CHAR(20),
   address2             CHAR(20),
   city                 CHAR(15),
   state                CHAR(2),
   zipcode              CHAR(5),
   phone                CHAR(18)
   );

CREATE INDEX c_temp1 ON customer (customer_num DESC);
CREATE INDEX c_temp2 ON customer (customer_num, zipcode);
```

In this example, the **customer_num** column has a unique constraint placed on it. The first CREATE INDEX statement places an index sorted in descending order on the **customer_num** column. The second CREATE INDEX includes the **customer_num** column as part of a composite index. For more information on composite indexes, see Creating Composite Indexes on page 245.

Bidirectional Traversal of Indexes

If you do not specify the ASC or DESC keywords when you create an index on a single column, key values are stored in ascending order by default; but the bidirectional-traversal capability of the database server lets you create just one index on a column and use that index for queries that specify sorting of results in either ascending or descending order of the sort column.

Because of this capability, it does not matter whether you create a single-column index as an ascending or descending index. Whichever storage order you choose for an index, the database server can traverse that index in ascending or descending order when it processes queries.

If you create a composite index on a table, however, the ASC and DESC keywords might be required. For example, if you want to enter a SELECT statement whose ORDER BY clause sorts on multiple columns and sorts each column in a different order, and you want to use an index for this query, you need to create a composite index that corresponds to the ORDER BY columns. For example, suppose that you want to enter the following query:

```
SELECT stock_num, manu_code, description, unit_price
   FROM stock ORDER BY manu_code ASC, unit_price DESC;
```

This query sorts first in ascending order by the value of the **manu_code** column and then in descending order by the value of the **unit_price** column. To use an index for this query, you need to issue a CREATE INDEX statement that corresponds to the requirements of the ORDER BY clause. For example, you can enter either of the following statements to create the index:

```
CREATE INDEX stock_idx1 ON stock
   (manu_code ASC, unit_price DESC);
CREATE INDEX stock_idx2 ON stock
   (manu_code DESC, unit_price ASC);
```

The composite index that was used for this query (**stock_idx1** or **stock_idx2**) cannot be used for queries in which you specify the same sort direction for the two columns in the ORDER BY clause. For example, suppose that you want to enter the following queries:

```
SELECT stock_num, manu_code, description, unit_price
   FROM stock ORDER BY manu_code ASC, unit_price ASC;
SELECT stock_num, manu_code, description, unit_price
   FROM stock ORDER BY manu_code DESC, unit_price DESC;
```

If you want to use a composite index to improve the performance of these queries, you need to enter one of the following CREATE INDEX statements. You can use either one of the created indexes (**stock_idx3** or **stock_idx4**) to improve the performance of the preceding queries.

```
CREATE INDEX stock_idx3 ON stock
   (manu_code ASC, unit_price ASC);
CREATE INDEX stock_idx4 ON stock
   (manu_code DESC, unit_price DESC);
```

You can create no more than one ascending index and one descending index on a column. Because of the bidirectional-traversal capability of the database server, you only need to create one of the indexes. Creating both would achieve exactly the same results for an ascending or descending sort on the **stock_num** column.

After INSERT or DELETE operations are performed on an indexed table, the number of index entries can vary within a page, and the number of index pages that a table requires can depend on whether the index specifies ascending or descending order. For some load and DML operations, a descending single-column or multi-column index might cause the database server to allocate more index pages than an ascending index requires.

Restrictions on the Number of Indexes on a Set of Columns

You can create multiple indexes on a set of columns, provided that each index has a unique combination of ascending and descending columns. For example, to create all possible indexes on the **stock_num** and **manu_code** columns of the **stock** table, you could create four indexes:

- The **ix1** index on both columns in ascending order
- The **ix2** index on both columns in descending order
- The **ix3** index on **stock_num** in ascending order and on **manu_code** in descending order
- The **ix4** index on **stock_num** in descending order and on **manu_code** in ascending order

Because of the bidirectional-traversal capability of the database server, you do not need to create these four indexes. You only need to create two indexes:

- The **ix1** and **ix2** indexes achieve the same results for sorts in which the user specifies the same sort direction (ascending or descending) for both columns, so you only need one index of this pair.
- The **ix3** and **ix4** indexes achieve the same results for sorts in which the user specifies different sort directions for the two columns (ascending on the first column and descending on the second column or vice versa). Thus, you only need to create one index of this pair. (See also Bidirectional Traversal of Indexes on page 247.)

Informix® can also support multiple indexes on the same combination of ascending and descending columns, if each index has a different collating order; see SET COLLATION statement on page 775.

Using an Operator Class

An *operator class* is the set of operators associated with a secondary-access method for query optimization and building the index. You must specify an operator class when you create an index if either one of the following is true:

- No default operator class for the secondary-access method exists. (A user-defined access method can provide no default operator class.)
- You want to use an operator class that is different from the default operator class that the secondary-access method provides.

If you use an alternative access method, and if the access method has a default operator class, you can omit the operator class here; but if you do not specify an operator class and the secondary-access method does not have a default operator class, the database server returns an error. For more information, see Default Operator Classes on page 270. The following CREATE INDEX statement creates a B-tree index on the **cust_tab** table that uses the **abs_btree_ops** operator class for the **cust_num** key:

```
CREATE INDEX c_num1_ix ON cust_tab (cust_num abs_btree_ops);
```

HASH ON clause

Use the HASH ON clause of the CREATE INDEX statement to specify the columns and number of subtrees (buckets) for a forest of trees index.

HASH ON clause

" HASH ON(*column*)WITH*number*BUCKETS "

Element	Description	Restrictions	Syntax
column	The name of the column or columns on which you use the HASH ON clause to create a forest of trees index	The list must be a prefix list of the index columns used in the CREATE INDEX statement	Identifier on page 1231
number	The number of subtrees (buckets) to create for a forest of trees index	The number of buckets for a forest of trees index must range from 2 to the number of available index pages per dbspace	Integer Literals on page 1199

Usage

Forest of trees indexes are detached indexes. They cannot be attached indexes.

You can create forest of trees indexes on columns with base data types.

You cannot:

- Create forest of trees indexes on columns with complex data types, UDTs, or functional columns.
- Use the FILLFACTOR option of the CREATE INDEX statement when you create forest of trees indexes, because the indexes are built from top to bottom.
- Create clustered forest of trees indexes.
- Run the ALTER INDEX statement on forest of trees indexes.
- Use forest of trees indexes in queries that use aggregates, including minimum and maximum range values
- Perform range scans directly on the HASH ON columns of a forest of trees index.

  However, you can perform range scans on columns that are not listed in the HASH ON column list. For range scans on columns listed in HASH ON column list, you must create an additional B-tree index that contains the appropriate column list for the range scan. This additional B-tree index might have the same column list as the forest of trees index, plus or minus a column.
- Use a forest of trees index for an `OR` index path. The database server does not use forest of trees indexes for queries that have an `OR` predicate on the indexed columns.

When you create a forest of trees index, choose enough columns to create unique values.

Tip: Generally, the columns to choose depend on the number of duplicates for each column. For example, if the first column contains a small number of duplicates, the first two columns are sufficient for hashing if they do not contain a large number of duplicates. If the first two columns contain a majority of duplicates, then you need to also choose a third column.

The number of subtrees depends on your goal for the index. If your goal is:

- To reduce contention, initially create a forest of trees index with 2 subtrees per CPU VP. You might need more subtrees, depending on the number of rows in the table and how many duplicates exist.
- To reduce the number of levels in the B-tree:
  1. Run the oncheck -pT command.
  2. In the output, find the number of nodes at each level.
  3. Determine how many subtrees are required to achieve the desired depth for each tree in the index.

For example, suppose an index averages 100 keys per page, the index has 1M keys, and the tree looks like this:

Level 1 (root) 100 keys
Level 2 10K keys
Level 3 1M keys

To reduce the 3-level tree to 100 2-level trees, the index needs roughly 100 subtrees. To reduce the 3-level tree to 10K 1-level trees, the index needs roughly 10K subtrees.

Forest of tree pages can be sparser than traditional B-tree pages if too many or too few subtrees are used. When the pages are sparser, more pages occupy the buffer pool, and therefore, cause other tables to become less cached.

Examples

The following command creates a forest of trees index named `idx1` with 100 subtrees on column c1:

```
CREATE INDEX idx1 ON tab1(c1) HASH ON (c1) with 100 buckets;
```

The following command creates a forest of trees index named `idx2`. In the command, the prefix list for the HASH ON portion of the statement is `c1` and `c2`, which is a prefix list of the `c1`, `c2`, and `c3` columns used in the CREATE INDEX portion of the statement.

```
CREATE INDEX idx2 on tab2(c1, c2, c3) HASH ON (c1, c2) with 10 buckets;
```

The following command creates a forest of trees index for equality lookups on columns `c1` and `c2`:

```
CREATE INDEX idx3 on tab3(c1, c2) HASH ON (c1, c2) with 100 buckets;
```

The following command creates a B-tree index that is similar to the previous forest of trees index. This index is for range scans on columns `c1` and `c2`:

```
CREATE INDEX idx4 on tab4(c1, c2, c3);
```

Storage options

The storage options specify the distribution scheme of an index. You can use the IN clause to specify a storage space for the entire index, or you can use the FRAGMENT BY clause to fragment the index across multiple storage spaces.

(explicit id unique_344_Connect_42_stop001) unique_344_Connect_42_stop001(explicit id unique_344_Connect_42_stop002)

unique_344_Connect_42_stop002(explicit id unique_344_Connect_42_stop003) unique_344_Connect_42_stop003(explicit id unique_344_Connect_42_stop004)

unique_344_Connect_42_stop004(explicit id unique_344_Connect_42_stop005) unique_344_Connect_42_stop005

Storage Options

" { **IN** { { *dbspace* [] | *extspace* [] } | [] **TABLE** [] } | <FRAGMENT BY Clause for Indexes> [] } "

Element	Description	Restrictions	Syntax
dbspace	The dbspace in which to store the index	Must exist	Identifier on page 1231
extspace	Name assigned by the onspaces command to a storage area outside the database server	Must exist	See the documentation for your access method.

If you specify any storage option (except IN TABLE), you create a *detached index*. Detached indexes are indexes that are created with a specified distribution scheme. Even if the distribution scheme specified for the index is identical to that specified for the table, the index is still considered to be detached. If the distribution scheme of a table changes, all detached indexes continue to use the distribution scheme that the Storage Option clause specified.

If you do not include the Storage Option clause, by default an attached index is created in the same dbspaces as the corresponding table fragments. However, if automatic location is enabled, an index created on a round-robin table is

detached by default in a single fragment, located in a dbspace chosen by the server. You enable automatic location by setting the AUTOLOCATE configuration parameter or session environment option to a positive integer.

Important:

When you are defining an index on a table that was created with AUTOLOCATE enabled, if a round-robin storage distribution scheme was automatically defined for the table, you cannot use the IN TABLE keywords to specify nonfragmented storage for the index. That storage option is valid only for B-tree indexes on nonfragmented tables.

COMPRESSED option for indexes

Use the COMPRESSED keyword of the CREATE INDEX statement to compress the B-tree index if the index has 2000 or more keys.

You can create a compressed index on a fragmented or non-fragmented table.

You cannot create a compressed index that is also a cluster index. However, you can compress an existing cluster index by running an SQL administration API task() or admin() function with the **index compress** argument.

To be compressed, the index or fragment within the index must have at least 2000 keys. If you use the COMPRESSED option when you create an index that does not have enough keys, the database server does not compress the index or fragment when it creates the index. The index remains uncompressed even if new keys are added to it. If you want to compress the index, run an SQL administration API task() or admin() function with the **index compress** argument.

If a table does not have enough data to provide a large enough sample of index keys, the database server does not compress the index or fragment. Even the minimum required number of new keys are added to an existing index that is not compressed, the database server does not compress the index.

However, after an index is compressed, the database server compresses and inserts any new key added to the index.

The following example creates a compressed index named **cust3_ix** on the address column of the customer table.

```
CREATE INDEX cust3_ix ON customer (address) COMPRESSED
   EXTENT SIZE 32 NEXT SIZE 32;
```

The following example creates a unique, compressed index:

```
CREATE UNIQUE INDEX cust3_ix ON customer (address) COMPRESSED ;
```

Extent Size Options

The Extent Size options can define the size of storage extents allocated to the index.

Extent Size Options

" [EXTENT SIZE *first_kilobytes*] "

" [NEXT SIZE *next_kilobytes*] "

Element	Description	Restrictions	Syntax
first_kilobytes	Length in kilobytes of the first extent for the index	Must return a positive number; maximum is the chunk size, in kilobytes	Expression on page 1003
next_kilobytes	Length in kilobytes of each subsequent extent	Same as for *first_kilobytes*	Expression on page 1003

The minimum length of *first_kilobytes* (and of *next_kilobytes*) is four times the disk-page size on your system. For example, if you have a 2-kilobyte page system, the minimum length is 8 kilobytes.

If you need to revise the extent sizes of an index, you can modify the extent and next-extent sizes in the generated schema files of an unloaded table. For example, to make a database more efficient, you might drop an index, modify the extent sizes in the schema files, and then create a new index. For information about how to optimize extents, see your *HCL® Informix® Administrator's Guide*.

Only extent size values that you explicitly assign as extent sizes for the new index are stored in the system catalog. The value that you specify in the EXTENT SIZE option to the CREATE INDEX statement is stored in the **fextsize** column of the **sysindices** system catalog table, and the value that you specify in the NEXT SIZE option is stored in the **nextsize** column of the same table. If you omit these options, however, the database server stores a value of zero (`0`) in those system catalog columns, rather than the default value that it calculates and allocates for the first extent or the next extent of the index.

Example of an index defined with explicit extent sizes

The following program fragment creates a new table and defines two nonfragmented indexes on the table.

```
CREATE TABLE IF NOT EXISTS t (a INT, b INT);
CREATE INDEX  IF NOT EXISTS idx1 ON t(a) EXTENT SIZE 32 NEXT SIZE 32;
CREATE INDEX  IF NOT EXISTS idx2 ON t(b);
```

Here the definition of **idx1** specifies 32 kilobytes as explicit extent sizes. The second index, **idx2**, has default extent sizes that the system calculates. The two CREATE INDEX statements produce system catalog descriptions of these indexes that include these extent size entries:

- The **sysindices.fextent** and **sysindices.nextent** column values are each `32` for **idx1**.
- The **sysindices.fextent** and **sysindices.nextent** column values are each `0` for **idx2**.

Here the `0` values for **idx2** indicate that no explicit extent sizes were specified (rather than indicating that no storage space was allocated).

IN Clause

Use the IN clause to specify a storage space to hold the entire index. The storage space that you specify must already exist.

Storing an Index in a dbspace

Use the IN *dbspace* clause to specify the dbspace where you want your index to reside. When you use this clause with any option except the TABLE keyword, you create a detached index.

The IN *dbspace* clause allows you to isolate an index. For example, if the **customer** table is created in the **custdata** dbspace, but you want to create an index in a separate dbspace called **custind**, use the following statements:

```
CREATE TABLE customer
   . . .
   IN custdata EXTENT SIZE 16;

CREATE INDEX idx_cust ON customer (customer_num) IN custind;
```

Storing an Index Fragment in a Named Partition

Besides the option of storing a fragment of the index in a dbspace, Informix® supports storing named fragments of the index in one or more dbspaces. Unless you explicitly declare names for the fragments in the PARTITION BY or FRAGMENT BY clause, each fragment, by default, has the same name as the dbspace where it resides. This includes all fragmented tables and indexes migrated from earlier releases of Informix®.

Storing Data in an extspace

In general, use the *extspace* storage option in conjunction with the #unique_342. For more information, refer to the user documentation for your custom-access method.

Creating an Index with the IN TABLE Keywords

Specifying IN TABLE as the storage option creates an index where both the index and the data pages for its table are stored together in the same extents, and the dbspace distribution scheme for the index is the same as that of the table on which it was built.

Using IN TABLE as the storage option specifies the same storage design for non-fragmented B-tree indexes as enabling the **DEFAULT_ATTACH** environment variable, but both **DEFAULT_ATTACH** and the IN TABLE keywords are deprecated features.

The name of the **DEFAULT_ATTACH** environment variable preserves an obsolete definition of the term *attached index*. In current Informix® nomenclature, this term now designates an index whose data pages are stored in separate tablespaces and separate extents from the data pages of the table, but the index and its table share the same dbspace distribution scheme. For more information, see the description of **DEFAULT_ATTACH** in the *HCL® Informix® Guide to SQL: Reference*.

The following restrictions apply to the IN TABLE keywords as an index storage option:

- If the table on which you define the index is a fragmented table, Informix® issues errors -212 and -130 if you specify the IN TABLE option.
- You cannot apply the IN TABLE storage option to forest of trees indexes.
- This option does not support extensibility-related indexes, such as R-tree indexes, functional indexes, or indexes that modules provide.
- You cannot specify this storage option for any index that uses a collating order different from that of its table, nor different from what the **DB_LOCALE** setting specifies. For more information about the **DB_LOCALE** environment variable, see the *HCL® Informix® Guide to SQL: Reference*.
- You cannot apply the IN TABLE storage option to indexes on tables implicitly fragmented by the AUTOLOCATE configuration parameter or by the SET ENVIRONMENT AUTOLOCATE session environment option.

For example, suppose that automatic location is enabled for the current session when the following DDL statements are issued:

```
CREATE TABLE tab_autoloc (col1 INT);
CREATE INDEX ind_intab ON tab_autoloc (col1) IN TABLE;
```

Although no explicit storage option is specified for the **tab_autoloc** table, it has an implicit round-robin storage distribution, based on the AUTOLOCATE setting. Consequently, the CREATE INDEX statement fails, and no **ind_intab** index is registered in the system catalog, because in-table indexes cannot be created on fragmented tables.

Restriction: When you are defining an index on a table that was created with AUTOLOCATE enabled, if a round-robin storage distribution scheme was automatically defined for the table, you cannot use the IN TABLE keywords to specify nonfragmented storage for the index. That storage option is valid only for B-tree indexes on nonfragmented tables.

IBM® does not recommend use in new applications of the IN TABLE storage option, nor of the **DEFAULT_ATTACH** environment variable. Such indexes are a deprecated feature that might not be supported in some future release of Informix®.

FRAGMENT BY Clause for Indexes

Use the FRAGMENT BY clause to create a detached index and to define its fragmentation strategy across dbspaces or partitions.

This closely resembles the syntax of the FRAGMENT BY clause for tables, but the ROUND ROBIN keywords are not supported for index fragmentation. The PARTITION BY keywords are a synonym for the FRAGMENT BY keywords in this context.

(explicit id unique_369_Connect_42_fbcl005) unique_369_Connect_42_fbcl005(explicit id unique_369_Connect_42_fbcl006)

unique_369_Connect_42_fbcl006(explicit id unique_369_Connect_42_fbcl007) unique_369_Connect_42_fbcl007

FRAGMENT BY Clause for Indexes

" { **FRAGMENT** | **PARTITION** } **BY** "

" { **EXPRESSION** <Expression Fragment Clause>[] | **RANGE**(*fragment_key*) <Interval Fragment Clause>[] | **LIST**(*fragment_key*) <List Fragment Clause>[] } "

Element	Description	Restrictions	Syntax
dbspace	The *dbspace* to store the index fragment	You can specify no more than 2,048 *dbspaces*. All *dbspaces* that store the fragments must have the same page size.	Identifier on page 1231
fragment_key	Cast, column, or function expressions on an index column. Index is fragmented on the value of this expression.	Columns must be from the current table only.	Expression on page 1003

Here the IN keyword introduces the name of a storage space where an index fragment is to be stored. If you list multiple *dbspace* names after the IN keyword, use parentheses to delimit the *dbspace* list. All *dbspaces* that store the fragments must have the same page size. The parentheses around the list of fragment definitions that follow the EXPRESSION keyword are optional.

For indexes that use the same RANGE interval or LIST fragmentation strategy as their table, each fragment name that you declare after the PARTITION keyword must be the same as the identifier of the corresponding table fragment.

For attached indexes that are fragmented by a RANGE interval fragmentation strategy, if no existing table fragment is in the range of a new inserted row, the database server creates a new table fragment to store the new row, and declares a system-generated name for the new table fragment. If the table is indexed, and the index is fragmented by the same RANGE interval strategy as its table, the database server also creates a new index fragment. In this case, the index fragment has the same system-generated identifier as the corresponding table fragment. For more information about system-generated RANGE interval fragments, see #unique_87.

Storing an Index Fragment in a Named Partition

Besides the option of storing a fragment of the index in a dbspace, Informix® supports storing named fragments of the index in one or more dbspaces. Unless you explicitly declare names for the fragments in the PARTITION BY or FRAGMENT BY clause, each fragment, by default, has the same name as the dbspace where it resides. This includes all fragmented tables and indexes migrated from earlier releases of Informix®.

Restrictions on fragmentation expressions

The following restrictions apply to the expression:

- Each fragment expression can contain columns only from the current table, with data values only from a single row.
- The expression must return a BOOLEAN (true or false) value.
- No subqueries, aggregates, user-defined routines, nor references to fields of a ROW type column or sequence objects are valid.
- The built-in CURRENT, SYSDATE, TODAY, SITENAME, DBSERVERNAME, CURRENT_USER, or USER functions are not valid.
- The DEFAULT_ROLE and CURRENT_ROLE operators are not valid.

The restrictions listed above also apply to indexes that use a LIST fragmentation strategy.

Fragmentation of System Indexes

System indexes (such as those that implement referential constraints and unique constraints) utilize user-defined indexes if they exist. If no user-defined indexes can be utilized, system indexes remain nonfragmented, and are moved to the dbspace where the database was created.

To fragment a system index, create the fragmented index on the constraint columns, and then add the constraint using the ALTER TABLE statement.

Fragmentation of Unique Indexes

You can fragment unique indexes on a table that uses a round-robin or an expression-based distribution scheme, but any columns referenced in the fragment expression must be indexed columns. If your index fragmentation strategy violates this restriction, the CREATE INDEX statement fails, and work is rolled back.

Fragmentation of Indexes on Temporary Tables

You can fragment a unique index on a temporary table only if the underlying table uses an expression-based distribution scheme. That is, the CREATE TEMP TABLE statement that defines the temporary table must specify an explicit expression-based distribution scheme. (Fragmentation of the index by ROUND ROBIN is not supported, and fragmentation by LIST or by INTERVAL is automatic, for a unique index on a table that uses a list or interval storage partitioning strategy.)

If you try to create a fragmented, unique index on a temporary table for which you did not specify a fragmentation strategy when you created the table, the database server creates the index in the first dbspace that the **DBSPACETEMP** environment variable specifies. For more information on the **DBSPACETEMP** environment variable, see the *HCL® Informix® Guide to SQL: Reference*.

For more information on the default storage characteristics of temporary tables, see Where temporary tables are stored on page 375.

Index modes

Use the index mode options of the CREATE INDEX statement to specify the behavior of the index during INSERT, DELETE, MERGE, and UPDATE operations.

Index Modes

```
" { ENABLED | [DISABLED] | FILTERING { WITHOUT ERROR | [WITH ERROR] } } "
```

DISABLED

The database server does not update the index after insert, delete, and update operations that modify the base table. The optimizer does not use the index during the execution of queries.

ENABLED

The database server updates the index after insert, delete, and update operations that modify the base table. The optimizer uses the index during query execution. If an insert or update operation causes a duplicate key value to be added to a unique index, the statement fails.

FILTERING

The database server updates a unique index after insert, delete, and update operations that modify the base table. (This option is not available with duplicate indexes.)

The optimizer uses the index during query execution. If an insert or update operation causes a duplicate key value to be added to a unique index in filtering mode, the statement continues processing, but the bad row is written to the violations table associated with the base table. Diagnostic information about the unique-index violation is written to the diagnostics table associated with the base table.

If you specify filtering for a unique index, you can also specify one of the following error options.

WITHOUT ERROR

A unique-index violation during an insert or update operation returns no integrity-violation error to the user.

WITH ERROR

Any unique-index violation during an insert or update operation returns an integrity-violation error to the user.

For information on changing the database object mode of a unique index, see Modes for constraints and unique indexes on page 788.

Specifying Modes for Unique Indexes

You must observe the following guidelines when you specify modes for unique indexes in CREATE INDEX statements:

- You can set the mode of a unique index to enabled, disabled, or filtering.
- If you do not specify a mode, then by default the index is enabled.
- For an index set to filtering mode, if you do not specify an error option, the default is WITHOUT ERROR.

- When you add a new unique index to an existing base table and specify the disabled mode for the index, your CREATE INDEX statement succeeds even if duplicate values in the indexed column would cause a unique-index violation.
- When you add a new unique index to an existing base table and specify the enabled or filtering mode for the index, your CREATE INDEX statement succeeds provided that no duplicate values exist in the indexed column that would cause a unique-index violation. However, if any duplicate values exist in the indexed column, your CREATE INDEX statement fails and returns an error.
- When you add a new unique index to an existing base table in the enabled or filtering mode, and duplicate values exist in the indexed column, erroneous rows in the base table are not filtered to the violations table. Thus, you cannot use a violations table to detect the erroneous rows in the base table.

Adding a Unique Index When Duplicate Values Exist in the Column

If you attempt to add a unique index in the enabled mode but receive an error message because duplicate values are in the indexed column, take the following steps to add the index successfully:

1. Add the index in the disabled mode. Issue the CREATE INDEX statement again, but this time specify the DISABLED keyword.
2. Start a violations and diagnostics table for the target table with the START VIOLATIONS TABLE statement.
3. Issue a SET Database Object Mode statement to change the mode of the index to enabled. When you issue this statement, existing rows in the target table that violate the unique-index requirement are duplicated in the violations table. You receive an integrity-violation error message, however, and the index remains disabled.
4. Issue a SELECT statement on the violations table to retrieve the nonconforming rows that are duplicated from the target table. You might need to join the violations and diagnostics tables to get all the necessary information.
5. Take corrective action on the rows in the target table that violate the unique-index requirement.
6. After you fix all the nonconforming rows in the target table, issue the SET Database Object Mode statement again to switch the disabled index to the enabled mode. This time the index is enabled, and no integrity violation error message is returned because all rows in the target table now satisfy the new unique-index requirement.

Specifying Modes for Duplicate Indexes

You must observe the following guidelines when you specify modes for duplicate indexes in CREATE INDEX statements:

- You can set a duplicate index to enabled or disabled mode. Filtering mode is available only for unique indexes.
- If you do not specify the mode of a duplicate index, by default the index is enabled.

How the Database Server Treats Disabled Indexes

Whether a disabled index is a unique or duplicate index, the database server effectively ignores the index during data-manipulation (DML) operations.

When an index is disabled, the database server stops updating it and stops using it during queries, but the catalog information about the disabled index is retained. You cannot create a new index on a column or set of columns if a disabled index on that column or set of columns already exists. Similarly, you cannot create an active (enabled) unique, foreign-key,

or primary-key constraint on a column or on a set of columns if the indexes on which the active constraint depends are disabled.

The ONLINE keyword of CREATE INDEX

The DBA can reduce the risk of nonexclusive access errors, and can increase the availability of the indexed table, by including the ONLINE keyword as the last specification of the CREATE INDEX statement. The ONLINE keyword instructs the database server to create the index while minimizing the duration of an exclusive lock, so that the index can be created while concurrent users are accessing the table.

By default, CREATE INDEX attempts to place an exclusive lock on the indexed table to prevent all other users from accessing the table while the index is being created. The CREATE INDEX statement fails if another user already has a lock on the table, or is currently accessing the table at the Dirty Read isolation level.

The database server builds the index, even if other users are performing Dirty Read and DML operations on the indexed table. Immediately after you issue the CREATE INDEX ONLINE statement, the new index is not yet visible to the query optimizer for use in query plans or cost estimates, and the database server does not support any other DDL operations on the indexed table, until after the specified index has been built without errors. At this time, the database server briefly locks the table while updating the system catalog with information about the new index.

The indexed table in a CREATE INDEX ONLINE statement can be permanent or temporary, logged or unlogged, and fragmented or non-fragmented. You cannot specify the ONLINE keyword, however, when you are creating an index that has any of the following attributes:

- a functional index
- a clustered index
- a virtual index
- an R-tree index
- an index that is partitioned by an interval fragmentation strategy
- an index on a table that is partitioned by an interval fragmentation strategy.

In addition, if a primary key constraint is defined on the table, a CREATE INDEX ONLINE operation can generate error -710 if one or more concurrent sessions are performing DML operations on a child table that has a foreign key constraint referencing that primary key. Before the index can be created ONLINE, you must wait until all the user sessions with those child tables have completed.

The following statement instructs the database server to create a unique online index called **idx_1** on the **lname** column of the **customer** table:

```
CREATE UNIQUE INDEX IF NOT EXISTS idx_1 ON customer(lname) ONLINE;
```

If, while this index is being constructed, other users insert into the **customer** table new rows in which **lname** is not unique, the database server issues an error after it has created the new **idx_1** index and registered it in the system catalog.

The term *online index* refers to the locking strategy that the database follows in creating or dropping an index with the ONLINE keyword, rather than to properties of the index that persist after its creation (or its destruction) has completed. This

term appears in some error messages, however, and in recovery or restore operations, the database server re-creates as an online index any index that you created as an online index.

No more than one CREATE INDEX ONLINE or DROP INDEX ONLINE statement can concurrently reference online indexes on the same table, or online indexes that have the same identifier.

For indexes defined with the IN TABLE keyword option, however, the indexed table remains locked for the duration of the CREATE INDEX ONLINE operation. While the in-table index is being created, attempted access by other sessions to the locked table would fail with these errors:

```
107: ISAM error:  record is locked.
211: Cannot read system catalog (systables).
710: Table (table.tix) has been dropped, altered or renamed.
```

Automatic Calculation of Distribution Statistics

When the CREATE INDEX statement runs successfully, with or without the ONLINE keyword, Informix® automatically gathers statistics for the newly created index, and updates the **sysdistrib** system catalog table with values that are equivalent to an UPDATE STATISTICS operation in a mode that depends on the type of index:

- Index level statistics, equivalent to the statistics gathered by UPDATE STATISTICS in the LOW mode, are calculated for most types of indexes, including B-tree, Virtual Index Interface, and functional indexes.
- Column distribution statistics, equivalent to the distribution generated in the HIGH mode, for a non-opaque leading indexed column of an ordinary B-tree index. The resolution percentage is 1.0 if the table has fewer than a million rows, and 0.5 for larger table sizes.

These distribution statistics are available to the query optimizer when it designs query plans for the table on which the new index was created.

For composite key indexes, only distributions of the leading column are created implicitly by the CREATE INDEX statement.

The implicit creation of distribution statistics is not supported for the following types of indexes:

- Indexes on columns of user-defined data types
- Indexes on columns of the built-in opaque data types (including BOOLEAN and LVARCHAR)
- R-tree indexes
- Attached indexes.

If the calculation of distribution statistics fails during the CREATE INDEX operation, the database server reports that failure in the error log, but continues to create the index.

When distributions are successfully created by an explicit or implicit CREATE INDEX operation, explain information (similar to one generated by UPDATE STATISTICS) such as following is generated if the SET EXPLAIN facility is set to ON.

```
Index:          idx_01 on nita.foo
STATISTICS CREATED AUTOMATICALLY:
Column Distribution for:               nita.foo.a
Mode:           MEDIUM
Number of Bins:     101 Bin size:   100.0
```

```
Sort data:           0.3 MB
Completed building distribution in:     0 minutes 33 seconds
```

See the description of the UPDATE STATISTICS statement on page 883 for information about distribution statistics and about the difference between LOW mode and MEDIUM mode distributions.

VISIBLE AND INVISIBLE options for indexes

Visibility option for indexes decides availability of the index to the optimizer during query optimization.

Index visibility Options

"[{ VISIBLE | INVISIBLE }]"

VISIBLE

An index that is set to be visible. A visible index is maintained and may be considered but not necessarily used by the optimizer. VISIBLE keyword is optional.

INVISIBLE

An index that is set to be invisible, is maintained but ignored by the optimizer.

The indexes are considered by the optimizer unless the indexes are disabled or directives are used. When the index is disabled, it is not maintained. Enabling a disabled index rebuilds the index.

A newly created index will be visible by default or can be set to visible using VISIBLE keyword while creating it. Use the INVISIBLE keyword of the CREATE INDEX statement to create an invisible index. The index will be created as invisible or can be made invisible from being visible.

Example

Example 1:

```
CREATE TABLE tab1(col1 int, col2 int, col3 int);
CREATE INDEX idx1 on tab1(col1);
CREATE INDEX idx2 on tab1(col2) VISIBLE;
CREATE INDEX idx3 on tab1(col3) INVISIBLE;
```

idx1 and idx2 will be created VISIBLE. Although idx1 is created without the VISIBLE keyword, by default it is VISIBLE. idx3 will be created INVISIBLE.

Dbschema/Dbexport and VISIBLE/INVISIBLE indexes

If the index is created invisible or made invisible using the SET INDEXES statement, dbschema and dbexport will append invisible keyword to the create index statement.

```
Using the indexes created in Example 1,  dbschema output will be:

After the create index statements:
create index "uaymak1".idx1 on "uaymak1".tab1 (col1) using btree ;
```

```
create index "uaymak1".idx2 on "uaymak1".tab1 (col2) using btree ;
create index "uaymak1".idx3 on "uaymak1".tab1 (col3) using btree invisible ;

Indexes are set from visible to invisible and vice versa:
set indexes "uaymak1".idx1 INVISIBLE ;
set indexes "uaymak1".idx2 INVISIBLE ;
set indexes "uaymak1".idx3 VISIBLE ;

After the set indexes statements:
create index "uaymak1".idx1 on "uaymak1".tab1 (col1) using btree invisible ;
create index "uaymak1".idx2 on "uaymak1".tab1 (col2) using btree invisible ;
create index "uaymak1".idx3 on "uaymak1".tab1 (col3) using btree ;
```

CREATE OPAQUE TYPE statement

Use the CREATE OPAQUE TYPE statement to create an opaque data type.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_384_Connect_42_creop002) unique_384_Connect_42_creop002 **CREATE OPAQUE TYPE [IF NOT EXISTS]** *type* **(INTERNALLENGTH =** { *length* **| VARIABLE** } [, <Opaque-Type Modifier>[]])

Element	Description	Restrictions	Syntax
length	Number of bytes needed to store a value of this data type	Positive integer returned when **sizeof( )** directive is applied to the type structure	Literal Number on page 1198
type	Name that you declare here for the new opaque data type	Must be unique among data type names in the database	Identifier on page 1231

Usage

The CREATE OPAQUE TYPE statement registers a new opaque data type in the **sysxtdtypes** system catalog table.

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if an OPAQUE data type of the specified name is already registered in the current database.

To create an opaque type, you must have the Resource privilege on the database. When you create the opaque type, only you, the owner, have the Usage privilege on the new opaque data type. You can use the GRANT or REVOKE statements to grant or revoke the Usage privilege of other users of the database.

To view the privileges on a data type, check the **sysxtdtypes** system catalog table for the owner name, and check the **sysxtdtypeauth** system catalog table for additional type privileges that might have been granted.

For details of system catalog tables, see the *HCL® Informix® Guide to SQL: Reference*.

The utility can also display privileges on opaque data types.

Declaring a Name for an Opaque Type

The name that you declare for an opaque data type is an SQL identifier. When you create an opaque type in a database that is not ANSI-compliant, the name must be unique among the names of data types within the database.

When you create an opaque type in an ANSI-compliant database, *owner*.*type* combination must be unique within the database. The owner name is case sensitive. If you do not put quotation marks around the owner name, the name of the opaque-type *owner* is stored in uppercase letters.

INTERNALLENGTH Modifier

The INTERNALLENGTH modifier specifies the storage size that is required for the opaque data type as fixed length or varying length.

Fixed-Length Opaque Types

A fixed-length opaque type has an internal structure of fixed size. To create a fixed-length opaque type, specify the size of the internal structure, in bytes, for the INTERNALLENGTH modifier. The next example creates a fixed-length opaque type called **fixlen_typ** and allocates 8 bytes for storing this data type.

```
CREATE OPAQUE TYPE fixlen_typ(INTERNALLENGTH=8, CANNOTHASH)
```

Varying-Length Opaque Types

A varying-length opaque type has an internal structure whose size might vary from one value to another. For example, the internal structure of an opaque data type might hold the actual value of a string up to a certain size, but beyond this size it might use an LO-pointer to a CLOB to hold the value.

To create a varying-length opaque data type, use the VARIABLE keyword with the INTERNALLENGTH modifier. The following statement creates a variable-length opaque data type called **varlen_typ**:

```
CREATE OPAQUE TYPE varlen_typ
(INTERNALLENGTH=VARIABLE, MAXLEN=1024)
```

Defining an Opaque Type

To define a new opaque data type to the database server, you must provide the following information in the C or Java™ language.

About this task

- A data structure that serves as the internal storage of the opaque data type

  The internal storage details of the type are hidden, or opaque. Once you define a new opaque data type, the database server can manipulate it without knowledge of the C or Java™ structure in which it is stored.
- Support functions that allow the database server to interact with this internal structure.

The support functions tell the database server how to interact with the internal structure of the data type. These support functions must be written in the C or Java™ programming language.

- Additional user-defined functions that other support functions or end users can invoke to operate on the opaque type (optional)

  Possible support functions include operator functions and cast functions. Before you can use these functions in SQL statements, they must be registered with the appropriate CREATE CAST, CREATE PROCEDURE, or CREATE FUNCTION statement.

The following table summarizes the support functions for an opaque data type.

Function	Description	Invoked
input( )	Converts the opaque type from its external LVARCHAR representation to its internal representation	When a client application sends a character representation of the opaque type in an INSERT, UPDATE, or LOAD statement
output( )	Converts the opaque type from its internal representation to its external LVARCHAR representation	When the database server sends a character representation of the opaque type as a result of a SELECT or FETCH statement
receive( )	Converts the opaque type from its internal representation on the client computer to its internal representation on the server computer Provides platform-independent results regardless of differences between client and server computer types	When a client application sends an internal representation of the opaque type in an INSERT, UPDATE, or LOAD statement
send( )	Converts the opaque type from its internal representation on the server computer to its internal representation on the client computer Provides platform-independent results regardless of differences between client and database server computer types	When the database server sends an internal representation of the opaque type as a result of a SELECT or FETCH statement
db_receive( )	Converts the opaque type from its internal representation on the local database to the **dbsendrecv** type for transfer to an external database on the local server	When a local database receives a **dbsendrecv** type from an external database on the local database server
db_send( )	Converts the opaque type from its internal representation on the local database to the **dbsendrecv** type for transfer to an external database on the local server	When a local database sends a **dbsendrecv** type to an external database on the local database server
server_receive( )	Converts the opaque type from its internal representation on the local server computer to the **srvsendrecv** type for	When the local database server receives a **srvsendrecv** type from a remote database server

Function	Description	Invoked
	transfer to a remote database server Use any name for this function.	
server_send( )	Converts the opaque type from its internal representation on the local server computer to the **srvsendrecv** type for transfer to a remote database server Use any name for this function.	When the local database server sends a **srvsendrecv** type to a remote database server
import( )	Performs any tasks needed to convert from the external (character) representation of an opaque type to the internal format for a bulk copy	When (LOAD) initiates a bulk copy from a text file to a database
export ( )	Performs any tasks needed to convert from the internal representation of an opaque type to the external (character) format for a bulk copy	When (UNLOAD) initiates a bulk copy from a database to a text file
importbinary( )	Performs any tasks needed to convert from the internal representation of an opaque type on the client computer to the internal representation on the server computer for a bulk copy	When (LOAD) or the High Performance Loader initiates a bulk copy from a binary file to a database
exportbinary( )	Performs any tasks needed to convert from the internal representation of an opaque type on the server computer to the internal representation on the client computer for a bulk copy	When (UNLOAD) or the High Performance Loader initiates a bulk copy from a database to a binary file
assign( )	Performs any processing required before storing the opaque type to disk This support function must be named **assign( )**.	When the database server executes INSERT, UPDATE, or LOAD, before it stores the opaque type to disk
destroy( )	Performs any processing necessary before removing a row that contains the opaque type This support function must be named **destroy( ).**	When the database server executes the DELETE or DROP TABLE, before it removes the opaque type from disk
lohandles( )	Returns a list of the LO-pointer structures (pointers to smart large objects) in an opaque type	When the database server must search opaque types for references to smart large objects; when **oncheck** runs, or an archive is performed
compare( )	Compares two values of the opaque type and returns an integer value to indicate whether the first value is less than, equal to, or greater than the second value	When the database server encounters an ORDER BY, UNIQUE, DISTINCT, or UNION clause in a SELECT statement, or when CREATE INDEX creates a B-tree index

After you write the necessary support functions for the opaque type, use the CREATE FUNCTION statement to register these support functions in the same database as the opaque type. Certain support functions convert other data types to or from

the new opaque type. After you create and register these support functions, use the CREATE CAST statement to associate each function with a particular cast. The cast must be registered in the same database as the support function.

After you have written the necessary C language or Java™ language source code to define an opaque data type, you then use the CREATE OPAQUE TYPE statement to register the opaque data type in the database.

CREATE OPCLASS statement

Use the CREATE OPCLASS statement to create an *operator class* for a *secondary-access method*.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_391_Connect_42_cropc002) unique_391_Connect_42_cropc002 **CREATE OPCLASS [IF NOT EXISTS]** *opclass*

FOR*sec_acc_method* **STRATEGIES(** <Strategy Specification>[] **) SUPPORT(** *support_function* **)**

Element	Description	Restrictions	Syntax
opclass	Name that you declare here for a new operator class	Must be unique among operator classes within the database	Identifier on page 1231
sec_acc_method	Secondary-access method with which the new operator class is associated	Must already exist and must be registered in the **sysams** table	Identifier on page 1231
support_function	Support function that the secondary-access method requires	Must be listed in the order expected by the access method	Identifier on page 1231

Usage

An *operator class* is the set of operators that support a secondary-access method for query optimization and building the index. A *secondary-access method* (sometimes referred to as an *index access method*) is a set of database server functions that build, access, and manipulate an index structure such as a B-tree, R-tree, or an index structure that a module provides.

The database server provides the B-tree and R-tree secondary-access methods. For more information on the **btree** secondary-access method, see Default Operator Classes on page 270.

Define a new operator class when you want one of the following:

- An index to use a different order for the data than the sequence that the default operator class provides
- A set of operators that is different from any existing operator classes that are associated with a particular secondary-access method

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if an operator class of the specified name is already registered in the current database.

You must have the Resource privilege or be the DBA to create an operator class. The actual name of an operator class is an SQL identifier. When you create an operator class, the *opclass* name must be unique within the database.

When you create an operator class in an ANSI-compliant database, the *owner.opclass* combination must be unique within the database. The owner name is case sensitive. If you do not put quotation marks around the *owner* name (or else set the **ANSIOWNER** environment variable), the name of the operator-class owner is stored in uppercase letters.

The following CREATE OPCLASS statement creates a new operator class called **abs_btree_ops** for the **btree** secondary-access method:

```
CREATE OPCLASS abs_btree_ops FOR btree
   STRATEGIES (abs_lt, abs_lte, abs_eq, abs_gte, abs_gt)
   SUPPORT (abs_cmp);
```

An operator class has two kinds of operator-class functions:

- Strategy functions

  Specify strategy functions of an operator class in the STRATEGY clause of the CREATE OPCLASS statement. In the preceding CREATE OPCLASS code example, the **abs_btree_ops** operator class has five strategy functions.

- Support functions

  Specify support functions of an operator class in the SUPPORT clause. In the preceding CREATE OPCLASS code example, the **abs_btree_ops** operator class has one support function.

STRATEGIES Clause

Strategy functions are functions that users can invoke within a DML statement to operate on a specific data type. The query optimizer uses the strategy functions to determine whether a given index can be used to process a query.

If a query includes a UDF or a column on which an index exists, and if the qualifying operator in the query matches any function in the STRATEGIES clause, then the query optimizer considers using the index for the query. For more information on query plans, see your .

When you create a new operator class, the STRATEGIES clause identifies the strategy functions for the secondary-access method. Each strategy specification lists the name of a strategy function (and optionally, the data types of its parameters). You must list these functions in the order that the secondary-access method expects. For the specific order of strategy operators for the default operator classes for a B-tree index and for an R-tree index, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

Strategy Specification

The STRATEGIES keyword introduces a comma-separated list of function names or function signatures for the new operator class. Each element of this list is called a *strategy specification* and has the following syntax:

Strategy Specification

" *strategy_function* "

" [(*input_type* , *input_type* [, *output_type*])] "

Element	Description	Restrictions	Syntax
input_type	Data type of an input parameter to the strategy function for which you intend to use a specific secondary-access method	A *strategy function* accepts two input parameters and can have one optional output parameter	Data Type on page 985
output_type	Data type of the optional output parameter of the strategy function	Optional output parameter for side-effect indexes	Data Type on page 985
strategy_function	Strategy function to associate with the specified operator class	Must be listed in the order that the specified secondary-access method expects	Identifier on page 1231

Each *strategy_function* is an external function. The CREATE OPCLASS statement does not verify that a user-defined function of the name you specify exists. However, for the secondary-access method to use the strategy function, the external function must be:

- Compiled in a shared library
- Registered in the database with the CREATE FUNCTION statement

Optionally, you can specify the signature of a strategy function in addition to its name. A strategy function requires two input parameters and an optional output parameter. To specify the function signature, specify:

- An *input data type* for each of the two input parameters of the strategy function, in the order that the strategy function uses them
- Optionally, one *output data type* for an output parameter of the strategy function

You can specify UDTs as well as built-in data types. If you do not specify the function signature, the database server assumes that each strategy function takes two arguments of the same data type and returns a BOOLEAN value.

Indexes on Side-Effect Data

Side-effect data are additional values that a strategy function returns after a query that contains the strategy function. For example, an image module might use a *fuzzy* index to search image data. The index ranks the images according to how closely they match the search criteria. The database server returns the rank values as side-effect data with the qualifying images.

SUPPORT Clause

Support functions are functions that the secondary-access method uses internally to build and search the index. Specify these functions for the secondary-access method in the SUPPORT clause of the CREATE OPCLASS statement.

You must list the names of the support functions in the order that the secondary-access method expects. For the specific order of support operators for the default operator classes for a B-tree index and an R-tree index, refer to Default Operator Classes on page 270.

The support function is an external function. CREATE OPCLASS does not verify that a specified support function exists. For the secondary-access method to use a support function, however, the support function must meet these criteria:

- Be compiled in a shared library
- Be registered in the database with the CREATE FUNCTION statement

Default Operator Classes

Each secondary-access method has a default operator class that is associated with it. By default, the CREATE INDEX statement associates the default operator class with an index.

For example, the following CREATE INDEX statement creates a B-tree index on the **zipcode** column and automatically associates the default B-tree operator class with this column:

```
CREATE INDEX zip_ix ON customer(zipcode)
```

For each of the secondary-access methods that Informix® provides, it provides a default operator class, as follows:

- The default B-tree operator class is a built-in operator class.

  The database server implements the operator-class functions for this operator class and registers it as **btree_ops** in the system catalog tables of a database.
- The default R-tree operator class is a registered operator class.

  The database server registers this operator class as **rtree_ops** in the system catalog tables. The database server does *not* implement the operator-class functions for the default R-tree operator class.



Important: To use an R-tree index, you must install a spatial module, such as a third-party module that implements the R-tree index. These implement the R-tree operator-class functions.

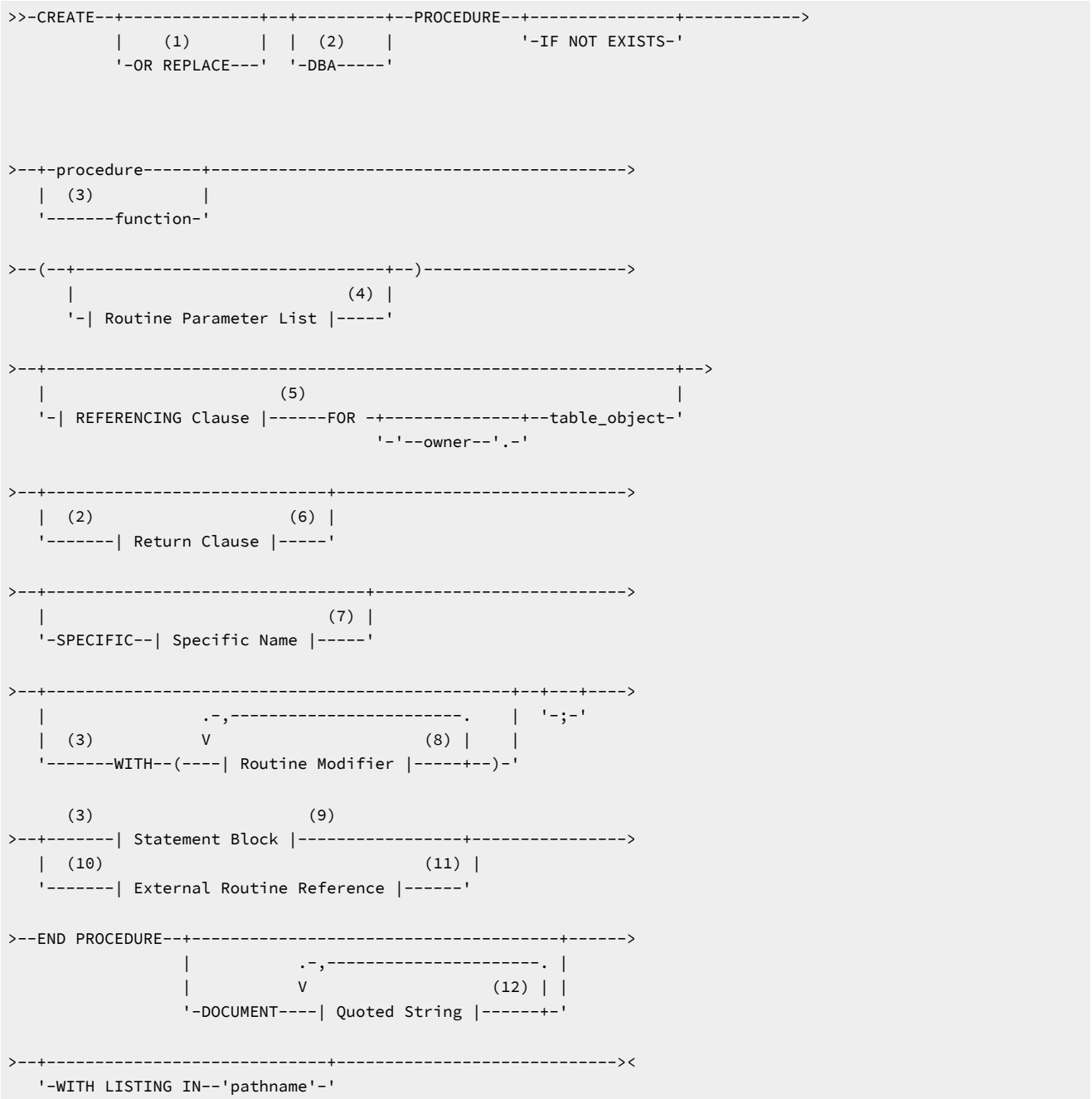
modules can provide other types of secondary-access methods. If a module provides a secondary-access method, it might also provide a default operator class. For more information, refer to your module user's guide.

CREATE PROCEDURE statement

Use the CREATE PROCEDURE statement to create a user-defined procedure. (To create a procedure from text of source code that is in a separate file, use the CREATE PROCEDURE FROM statement.)

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax



Note:

1. See OR REPLACE Clause on page 274
2. See DBA Keyword and Privileges on the Procedure on page 275
3. Stored Procedure Language only

4. See Routine Parameter List on page 1281
5. See The REFERENCING and FOR Clauses on page 276
6. See Return Clause on page 1271
7. See Specific Name on page 1288
8. See Routine modifier on page 1275
9. See Statement Block on page 1289
10. External routines only
11. See External Routine Reference on page 1229
12. See Quoted String on page 1202

Element	Description	Restrictions	Syntax
function, *procedure*	Name declared here for a new SPL routine	See Procedure names in Informix on page 278.	Identifier on page 1231
owner	Owner of *table_object*	Must own *table_object*	Owner name on page 1262
pathname	File to store compile-time warnings	Must exist on the computer where the database resides	Operating system specific
table_object	Name or synonym of a table or view whose triggers can call this UDR	Must exist in the local database	Identifier on page 1231

Usage

In , you can use CREATE PROCEDURE only as text within a PREPARE statement. If you want to create a procedure for which the text is known at compile time, you must use a CREATE PROCEDURE FROM statement.

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if a procedure of the specified name is already registered in the current database. (Because the identifier of a procedure can be overloaded, it might be unnecessary to include these keywords, if the database server can resolve the argument list of the new procedure as different from that of any other procedure of the same name in the current database.)

Routines use the collating order that was in effect when they were created. See SET COLLATION statement on page 775 statement of Informix® for information about using non-default collation.

Example

For this example, assume that you have two overloaded procedures that are defined as follows:

```
CREATE PROCEDURE raise_prices ( per_cent INT)
 UPDATE stock SET unit_price = unit_price + (unit_price * (per_cent/100) );
END PROCEDURE

CREATE PROCEDURE raise_prices ( per_cent INT, selected_unit CHAR )
```

```
 UPDATE stock SET unit_price = unit_price + (unit_price * (per_cent/100) )
 where unit=selected_unit;
END PROCEDURE
```

In order to refer to the above procedures, you would need to provide the procedure name followed by the parameter list, as in the following examples:

```
DROP PROCEDURE raise_prices(INT);
DROP PROCEDURE raise_prices(INT, CHAR);
```

A more convenient way is to use the specific name to identify each of them. The following example will create the procedure using the specific name:

```
CREATE PROCEDURE raise_prices ( per_cent INT ) SPECIFIC
  raise_prices_all
 UPDATE stock SET unit_price = unit_price + (unit_price * (per_cent/100) );
END PROCEDURE

DROP SPECIFIC PROCEDURE raise_prices_all;

CREATE PROCEDURE raise_prices ( per_cent INT, selected_unit CHAR )
  SPECIFIC raise_prices_by_unit
 UPDATE stock SET unit_price = unit_price + (unit_price * (per_cent/100) )
 where unit=selected_unit;
END PROCEDURE
```

We can simply drop them using their specific names:

```
DROP SPECIFIC PROCEDURE raise_prices_by_all;
DROP SPECIFIC PROCEDURE raise_prices_by_unit;
```

Using CREATE PROCEDURE Versus CREATE FUNCTION

Although you can use CREATE PROCEDURE to write and register an SPL routine that returns one or more values (that is, an SPL function)In Informix®, it is recommended that you use CREATE FUNCTION instead. To register an external function, you must use CREATE FUNCTION.

Use the CREATE PROCEDURE statement to write and register an SPL procedure or to register an external procedure.

For information on how terms such as user-defined procedures and user-defined functions are used in this document, see Relationship Between Routines, Functions, and Procedures on page 273.

Relationship Between Routines, Functions, and Procedures

A *procedure* is a routine that can accept arguments but does not return any values. A *function* is a routine that can accept arguments and returns one or more values. *User-defined routine* (UDR*)* is a generic term that includes both user-defined procedures and user-defined functions. For information about named and unnamed returned values, see Return Clause on page 1271.

You can write a UDR in SPL (a SPL *routine*) or in an external language (an *external routine*) that the database server supports. Where the term UDR appears in this document, it can refer to both SPL routines and external routines.

The term *user-defined procedure* refers to SPL procedures and external procedures. *User-defined function* refers to SPL functions and external functions.

In the documentation of earlier releases, the term *stored procedure* was used for both SPL procedures and SPL functions. In this document, the term *SPL routine* replaces the term stored procedure. When it is necessary to distinguish between an SPL function and an SPL procedure, this document does so.

The term *external routine* applies to an external procedure or an external function, both constructs designating UDRs that are written in a programming language other than SPL. When it is necessary to distinguish between an external function and an external procedure, this document does so.

Privileges Necessary for Using CREATE PROCEDURE

You must have the Resource privilege on a database to create a user-defined procedure within that database.

Before you can create an SPL procedure, you must also have the Usage privilege on the SPL, C, or Java™ language in which the procedure is written. For more information, see Language-Level Privileges on page 549.

By default, the Usage privilege on SPL is granted to PUBLIC. You must also have at least the Resource privilege on a database to create an SPL procedure within that database.

OR REPLACE Clause

The OR REPLACE clause is used to update an existing procedure without changing any object privileges granted on it.

Example

```
create procedure testproc() returning char(50);
    return "this is the original procedure";
end procedure;
Routine created.

call testproc();

(expression)

this is the original procedure

1 row(s) retrieved.

create procedure testproc() returning char(50);
    return "this is the replaced procedure";
end procedure;
  673: Another routine (testproc) with same signature already exists in database


create or replace procedure testproc() returning char(50);
    return "this is the replaced procedure";
end procedure;
Routine created.

call testproc();
```

```
(expression)

this is the replaced procedure

1 row(s) retrieved.
```

DBA Keyword and Privileges on the Procedure

If you create a UDR with the DBA keyword, it is known as a DBA-privileged UDR. You need the DBA privilege to create a DBA-privileged UDR. If you omit the DBA keyword, the UDR is known as an owner-privileged UDR. Other users who have Execute privilege on the UDR run the UDR with the privileges of the user who created the UDR.

DBA-privileged UDRs

For a DBA-privileged UDR, users who do not hold the DBA privilege can run the routine under the following conditions:

- The DBA grants the user Execute privilege on the UDR.
- The DBA grants the Execute privilege to PUBLIC so that all users can run the UDR.

When you have Execute privilege on a DBA-privileged UDR, you run the UDR with DBA privileges.

For additional information about DBA-privileged UDRs, see Ownership of Created Database Objects on page 281.

Owner-privileged UDRs

In an ANSI-compliant database, before other users can execute an owner-privileged UDR, the owner must grant the Execute privilege, either to individual users, or to roles, or to PUBLIC.

In a database that is not ANSI compliant, any user can run the UDR because PUBLIC is granted the Execute privilege by default. To restrict access to an owner-privileged UDR to specific users, the owner must revoke the Execute privilege on the UDR from PUBLIC, and then grant it to specified users or roles. However, you can prevent privileges on UDRs from being granted to PUBLIC by default. by setting the **NODEFDAC** environment variable to `yes`. Then the owner must grant the Execute privilege for the UDR to other users.

For an owner-privileged UDR, all non-qualified database objects that the UDR references are implicitly qualified by the name of the UDR owner. To allow other users to run the UDR, the owner of the UDR must be granted the necessary privileges on any underlying database objects with the WITH GRANT OPTION keyword in the GRANT statement.

The privileges of the owner determine who can run an owner-privileged UDR:

- If the owner does not have privileges on all of the objects that are referenced by the UDR, then the UDR fails regardless of whether the owner or another user runs it.
- If the owner has privileges on all referenced objects, but does not have the GRANT option, then only the owner can run the UDR. Other users cannot run the UDR, even if they have Execute privilege on the UDR.
- If the owner has privileges with the GRANT option on all referenced objects, then the owner and other users who have the Execute privilege can run the UDR. Other users who have Execute privilege run the UDR with the owner's privileges.

The REFERENCING and FOR Clauses

The REFERENCING clause can declare correlation names for the original value and for the updated value in columns of the *table_object* that the FOR clause specifies

(explicit id unique_308_Connect_42_r4crp002) unique_308_Connect_42_r4crp002

REFERENCING and FOR Clauses

" **REFERENCING** "

" { **OLD** | **NEW** } [**AS**] *correlation* "

" **FOR** [*'owner'* .] *table_object* "

Element	Description	Restrictions	Syntax
correlation	Name that you declare here to qualify an old or new column value (as *correlation*.*column*) in a trigger routine	Must not be *table_object*	Identifier on page 1231
owner	Owner of *table_object*	Must own *table_object*	Owner name on page 1262
table_object	Name or synonym for the table or view whose triggers can call this procedure	Must exist in the local database	Identifier on page 1231

If you include the REFERENCING and FOR *table_object* clauses immediately after the parameter list of the CREATE PROCEDURE statement, the routine that you create is known as a *trigger procedure* (or *trigger UDR* or *trigger routine*). The FOR clause specifies the table or view whose triggers can invoke the routine from the FOR EACH ROW section of their Triggered Action list.

In the REFERENCING clause, the OLD *correlation* specifies a prefix by which the trigger routine can reference the value that a column of *table_object* had before the trigger routine modifies that column value. The NEW *correlation* specifies a prefix for referencing the new value that the trigger routine assigns to the column. Whether the trigger routine can use correlation names to reference the OLD column value, the NEW column value, or both values depends on the type of triggering event:

- A trigger routine invoked by an Insert trigger can reference only the NEW correlation name.
- A trigger routine invoked by a Delete trigger or by a Select trigger can reference only the OLD correlation name.
- A trigger routine invoked by an Update trigger can reference both the OLD and the NEW correlation names.

For information about how to use the *correlation.column* notation in triggered actions, see REFERENCING Clauses on page 395.

Besides the general requirements for any Informix® UDR that is written in the SPL language, trigger routines can support certain additional syntax features, and are subject to certain restrictions, that are not features (or that are not restrictions) for ordinary UDRs that are not trigger routines:

- A trigger routine must include the FOR *table_object* clause that specifies the name of the table or view in the local database whose triggers can invoke this routine.
- A trigger routine can also include the REFERENCING clause to declare correlation names for OLD and NEW values that SPL statements in the UDR can reference.
- Trigger routines can be invoked only in the FOR EACH ROW section of the Triggered Action list in the trigger definition.
- Correlated variables for OLD or NEW values can appear in the IF statement of SPL and in CASE expressions.
- Correlated variables for OLD values cannot be on the left-hand side of a LET expression
- Correlated variables for NEW values cannot be on the left-hand side of a LET expression if the FOR clause specifies a view whose INSTEAD OF trigger action list invokes the trigger routine.
- Only correlated variables for NEW values can be on the left-hand side of a LET expression that references correlated variables. In this case, the FOR clause must specify a table, rather than a view, and the trigger whose action invokes the SPL routine cannot be an INSTEAD OF trigger.
- Both OLD and NEW values can be on the right-hand side of a LET expression.
- The Boolean operators SELECTING, INSERTING, DELETING, and UPDATING are valid in trigger routines (and only in trigger routines and in other UDRs that are invoked in triggered action statements) in contexts where Boolean expressions are valid. These operators return TRUE ('t') if the triggering event matches the DML operation referenced by the name of the operator, and they return FALSE ('f') otherwise.
- If a single triggering event activates multiple triggers on the same table or view, then all of the BEFORE actions take place before any of the FOR EACH ROW actions, and all of the AFTER actions follow the FOR EACH ROW actions. The order of execution of different triggers on the same event is not guaranteed.
- Trigger routines must be written in the SPL language. They cannot be written in an external language, like the C or Java™ language, but they can include calls to external language routines, such as the **mi_trigger** application programming interface for trigger introspection.
- Trigger routines cannot reference savepoints. Any changes to the data values or to the schema of the database by a triggered action must be committed or rolled back in their entirety. Partial rollback of a triggered action is not supported.

For more information about the **mi_trigger** API, refer to the *HCL® Informix® DataBlade® API Programmer's Guide* and to the *IBM® Informix® DataBlade® API Function Reference*.

If you include the REFERENCING clause but omit the FOR clause, or if you include the FOR clause but omit the REFERENCING clause, the CREATE PROCEDURE statement fails with an error.

If you omit the REFERENCING and FOR clauses, the UDR cannot use the SELECTING, INSERTING, DELETING, and UPDATING operators, and cannot declare variables that can represent and manipulate column values in triggered actions on the table or view that the trigger definition specifies.

See the REFERENCING Clauses on page 395 section in the CREATE TRIGGER statement description for the syntax of the REFERENCING clause for Delete, Insert, Select, and Update triggers on tables, and for Delete, Insert, and Update INSTEAD OF triggers on views.

Procedure names in Informix®

Because HCL Informix® offers *routine overloading*, you can define more than one user-defined routine (UDR) with the same name, but different parameter lists. You might want to overload UDRs in the following situations:

- You create a UDR with the same name as a built-in routine (such as **equal( )**) to process a new user-defined data type.
- You create *type hierarchies* in which subtypes inherit data representation and UDRs from supertypes.
- You create *distinct types*, which are data types that have the same internal storage representation as an existing data type, but have different names and cannot be compared to the source type without casting. Distinct types inherit UDRs from their source types.

For a brief description of the routine signature that uniquely identifies each UDR, see Routine Overloading and Routine Signatures on page 1228.

Using the SPECIFIC Clause to Specify a Specific Name

You can declare a *specific name* that is unique in the database for a user-defined procedure. A specific name is useful when you are overloading a procedure.

DOCUMENT Clause

The quoted string in the DOCUMENT clause provides a synopsis and description of a UDR. The string is stored in the **sysprocbody** system catalog table and is intended for the user of the UDR.

Anyone with access to the database can query the **sysprocbody** system catalog table to obtain a description of one or all the UDRs stored in the database. A UDR or application program can query the system catalog tables to fetch the DOCUMENT clause and display it for a user.

For example, to find the description of the SPL procedure **raise_prices**, shown in SPL Procedures on page 279, enter a query such as this example:

```
SELECT data FROM sysprocbody b, sysprocedures p
WHERE b.procid = p.procid
      --join between the two catalog tables
   AND p.procname = 'raise_prices'
      -- look for procedure named raise_prices
   AND b.datakey  = 'D';-- want user document
```

The preceding query returns the following text:

```
USAGE: EXECUTE PROCEDURE raise_prices( xxx )
xxx = percentage from 1 - 100
```

For external procedures, you can use a DOCUMENT clause at the end of the CREATE PROCEDURE statement, whether or not you use the END PROCEDURE keywords.

SPL Procedures

SPL procedures are UDRs written in Stored Procedure Language (SPL) that do not return a value. To write and register an SPL routine, use the CREATE PROCEDURE statement. Embed appropriate SQL and SPL statements between the CREATE PROCEDURE and END PROCEDURE keywords. You can also follow the UDR definition with the DOCUMENT and WITH FILE IN options.

SPL routines are parsed, optimized (as far as possible), and stored in the system catalog tables in executable format. The body of an SPL routine is stored in the **sysprocbody** system catalog table. Other information about the routine is stored in other system catalog tables, including **sysprocedures**, **sysprocplan**, and **sysprocauth**.

If the Statement Block portion of the CREATE PROCEDURE statement is empty, no operation takes place when you call the procedure. You might use such a "placeholder" procedure in the development stage when you intend to establish the existence of a procedure but have not yet coded it.

If you specify an optional clause after the parameter list, you must place a semicolon after the clause that immediately precedes the Statement Block.

The following example creates an SPL procedure:

```
CREATE PROCEDURE raise_prices ( per_cent INT )
   UPDATE stock SET unit_price =
      unit_price + (unit_price * (per_cent/100));
END PROCEDURE
   DOCUMENT "USAGE: EXECUTE PROCEDURE raise_prices( xxx )",
   "xxx = percentage from 1 - 100 "
   WITH LISTING IN '/tmp/warn_file';
```

External Procedures

About this task

External procedures are procedures you write in an external programming language that the database server supports. (Procedures written in the SPL language are not external procedures.)

To create a C user-defined procedure

1. Write a C function that does not return a value.
2. Compile the C function and store the compiled code in a shared library (the shared-object file for C).
3. Register the C function in the database server with the CREATE PROCEDURE statement.

Results

To create a user-defined procedure written in the Java™ language:

1. Write a Java™ static method, which can use the JDBC functions to interact with the database server.
2. Compile the Java™ source and create a JAR file (the shared-object file).
3. Execute the **install_jar( )** procedure with the EXECUTE PROCEDURE statement to install the JAR file in the current database.
4. If the UDR uses user-defined types, create a mapping between SQL data types and Java™ classes, using the **setUDTextName( )** procedure that is explained in EXECUTE PROCEDURE statement on page 510.
5. Register the UDR with the CREATE PROCEDURE statement. (If an external routine returns a value, you must register it with the CREATE FUNCTION statement, rather than with CREATE PROCEDURE.)

Rather than storing the body of an external routine directly in the database, the database server stores only the pathname of the shared-object file that contains the compiled version of the routine. The database server executes an external routine by invoking the external object code.

You must also hold either the Resource privilege or the DBA privilege on the database in which the external procedure will be registered, as well as the Usage privilege on the programming language in which the routine is written. (For the syntax of granting Usage privileges on the C language or on the Java™ language to a user, or to a role, or to the PUBLIC group, see Language-Level Privileges on page 549.)

When the IFX_EXTEND_ROLE configuration parameter is set to `1` or to `ON`, only users who have the built-in EXTEND role can create external procedures.

Registering a User-Defined Procedure

About this task

This example registers a C user-defined procedure named **check_owner( )** that takes one argument of the type LVARCHAR. The external routine reference specifies the path to the C shared library where the procedure object code is stored. This library contains a C function **unix_owner( )**, which is invoked during execution of the **check_owner( )** procedure.

```
CREATE PROCEDURE check_owner ( owner lvarchar )
   EXTERNAL NAME "/usr/lib/ext_lib/genlib.so(unix_owner)"
   LANGUAGE C
END PROCEDURE;
```

This example registers a user-defined procedure named **showusers( )** that is written in the Java™ language:

```
CREATE PROCEDURE showusers()
   WITH (CLASS = "jvp") EXTERNAL NAME 'admin_jar:admin.showusers'
   LANGUAGE JAVA;
```

The EXTERNAL NAME clause specifies that the Java™ implementation of the **showusers( )** procedure is a method called **showusers( )**, which resides in the **admin** Java™ class that resides in the **admin_jar** JAR file.

Ownership of Created Database Objects

The user who creates an owner-privileged UDR owns any database objects that the UDR creates when it executes, unless some other *owner* is specified for the object. In other words, the UDR owner, not the user who executes the owner-privileged UDR, is the owner of any database objects created by the UDR unless another owner is specified in the DDL statement that creates the database object.

In the case of a DBA-privileged UDR, however, the user who executes the UDR, not the UDR owner, owns any database objects that the UDR creates, unless some other owner is specified for the database object within the UDR.

For examples, see Ownership of Created Database Objects on page 235 in the description of the CREATE FUNCTION statement.

CREATE PROCEDURE FROM statement

Use the CREATE PROCEDURE FROM statement to access a user-defined procedure. The actual text of the CREATE PROCEDURE statement resides in a separate file.

This statement is an extension to the ANSI/ISO standard for SQL. You can use this statement with .

Syntax

```
CREATE PROCEDURE FROM [IF NOT EXISTS] { 'file' | file_var }
```

Element	Description	Restrictions	Syntax
file	Pathname and filename of file that contains full text of a CREATE PROCEDURE statement. Default pathname is the current directory.	Must exist, and can contain only one CREATE PROCEDURE statement. See also Default Directory That Holds the File on page 282.	Operating-system specific
file_var	Name of a program variable that contains *file* specification	Must be of a character data type; its contents have same restrictions as *file*	Language specific

Usage

You cannot create a user-defined procedure directly in programs. That is, the program cannot contain the CREATE PROCEDURE statement.

To use a user-defined procedure in an ESQL/C program:

1. Create a source file with the CREATE PROCEDURE statement.
2. Use the CREATE PROCEDURE FROM statement to send the contents of this source file to the database server for execution.

   The file can contain only one CREATE PROCEDURE statement.

For example, suppose that the following CREATE PROCEDURE statement is in a separate file, called **raise_pr.sql**:

```
CREATE PROCEDURE raise_prices( per_cent INT )
   UPDATE stock -- increase by percentage;
   SET unit_price = unit_price +
      ( unit_price * (per_cent / 100) );
END PROCEDURE;
```

In the program, you can access the **raise_prices( )** SPL procedure with the following CREATE PROCEDURE FROM statement:

```
EXEC SQL create procedure from 'raise_pr.sql';
```

If you are not sure whether the UDR in the file returns a value, use the CREATE ROUTINE FROM statement.

When the IFX_EXTEND_ROLE configuration parameter is set to ON, only users who have the built-in EXTEND role can create external routines.

When the IFX_EXTEND_ROLE configuration parameter is set to `1` or to `ON`, only users to whom the Database Server Administrator (DBSA) has granted the built-in EXTEND role can create external routines. In addition, you must hold at least the Resource access privilege on the database in which the routine will be registered. You must also hold the Usage privilege on the programming language in which the routine is written. (For the syntax of granting Usage privileges on the C language to a user or to a role, see Language-Level Privileges on page 549.)

User-defined procedures, like user-defined functions, use the collating order that was in effect when they were created. See SET COLLATION statement on page 775 for information about using non-default collation.

Default Directory That Holds the File

The database server treats the specified filename (and any pathname) as relative.

On UNIX™, if you specify a simple filename instead of a full pathname as the *file* parameter, the client application looks for the file in your home directory on the computer where the database resides. If you do not have a home directory on this computer, the default directory is the root directory.

On Windows™, if you specify a filename but no directory as the *file* parameter, the client application looks for the file in your current working directory if the database is on the local computer. Otherwise, the default directory is **%INFORMIXDIR%\bin**.

Important: The preprocessor does not process the contents of the file that you specify. It only sends the contents to the database server for execution. Therefore, there is no syntactic check that the file that you specify in CREATE PROCEDURE FROM actually contains a CREATE PROCEDURE statement. To improve readability of the code, however, it is recommended that you match these two statements.

CREATE ROLE statement

Use the CREATE ROLE statement to declare and register a new role.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
CREATE ROLE [IF NOT EXISTS] { role | 'role' }
```

Element	Description	Restrictions	Syntax
role	Name declared here for a role that the DBA creates	Must be unique among *role* and user names in the database. Maximum number of bytes is 32.	Owner name on page 1262

Usage

CREATE ROLE declares a new role and registers it in the system catalog. A role can associate a set of authorization identifiers with a set of access privileges on database objects. The system catalog maintains information about the roles (and their corresponding privileges) that are granted to users or to other roles.

Only the database administrator (DBA) can use CREATE ROLE to create a new role. The DBA can assign the privileges required for some work task to a role, such as **engineer**, and then use the GRANT statement to assign that role to specific users, instead of granting that set of privileges to each user individually.

The *role* name is an *authorization identifier*. It cannot be a user name that is known to the database server or to the operating system of the database server. The *role* name cannot already be listed in the **username** column of the **sysusers** system catalog table, nor in the **grantor** or **grantee** columns of the **systabauth**, **syscolauth**, **sysfragauth**, **sysprocauth**, or **sysroleauth** system catalog tables.

The *role* name also cannot match the name of any user or role that is already listed in the **grantor** or **grantee** columns of the **sysxtdtypeauth** system catalog table, nor any built-in role, such as EXTEND or DBSECADM.

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if a role of the specified name is already registered in the current database.

After a role is created, the DBA can use the GRANT statement to assign the role to PUBLIC, to users, or to other roles, and to grant specific privileges to the role. (A role cannot, however, hold database-level privileges.) After a role is granted successfully to a user or to PUBLIC, the user must use the SET ROLE statement to enable the role. Only then can the user exercise the privileges of the role.

To create the role **engineer**, for example, enter the following statement:

```
CREATE ROLE engineer;
```

To grant access privileges to the role **engineer**, the DBA can issue GRANT statements that include **engineer** in the list of grantees:

```
GRANT USAGE ON LANGUAGE SPL TO engineer;
```

To assign the role **engineer** to user **kaycee**, the DBA could issue this statement:

```
GRANT engineer TO kaycee;
```

To activate the role **engineer**, user **kaycee** must issue the following statement:

```
SET ROLE engineer;
```

If this SET ROLE statement is successful, user **kaycee** acquires whatever privileges have been granted to the role **engineer**, in addition to any other privileges that **kaycee** already holds as an individual or as PUBLIC.

A user can be granted several roles, but no more than one non-default role, as specified by SET ROLE, can be enabled for any user at a given time.

An exception to requiring SET ROLE to explicitly enable a role is any default role that the DBA specifies in the GRANT DEFAULT ROLE *role* TO *user* statement. If that statement succeeds, the default *role* is automatically enabled when *user* connects to the database. Any role can be a default role. (Similarly, users to whom the Informix® DBSA grants the EXTEND role need not execute SET ROLE before they can create and drop external routines and shared libraries.)

CREATE ROLE, when used with the GRANT and SET ROLE statements, enables a DBA to create one set of privileges for a role and then grant the role to many users, instead of granting the same set of privileges individually to many users.

With the GRANT DEFAULT ROLE and SET ROLE DEFAULT statements, *default roles* enable a DBA to assign privileges to a role that is activated automatically when any user who holds that default role connects to the database. This feature is useful when an application performs operations that require specific access privileges, but the application does not include SET ROLE statements.

The REVOKE statement can cancel access privileges of a role, remove users from a role, or cancel the default status of a role for one or more users. A role exists until either the DBA or a user to whom the role was granted with the WITH GRANT OPTION keywords uses the DROP ROLE statement to drop the role.

Important: The scope of a user-defined role (and of discretionary access privileges that the GRANT statement assigns to the role) is the current database. When the GRANT DEFAULT ROLE or SET ROLE statement activates a role, the role and its privileges take effect in the current database only. As a security precaution, discretionary access privileges that a user receives only from a role cannot provide access to tables outside the current database through a view or through a trigger action.

CREATE ROUTINE FROM statement

Use the CREATE ROUTINE FROM statement to register a UDR by referencing the text of a CREATE FUNCTION statement or CREATE PROCEDURE statement that resides in a separate file.

This statement is an extension to the ANSI/ISO standard for SQL.

You can use this statement with ESQL/C.

Syntax

```
CREATE ROUTINE FROM [IF NOT EXISTS] { 'file' | file_var }
```

Element	Description	Restrictions	Syntax
file	Pathname and filename for the text of a CREATE PROCEDURE or CREATE FUNCTION statement. Default path is the current directory.	Must exist and can contain only one CREATE FUNCTION or CREATE PROCEDURE statement.	Operating-system dependent
file_var	Name of a program variable that contains *file* specification	Must be a character data type; contents must satisfy *file* restrictions	Language specific

Usage

ESQL/C programs cannot use the CREATE FUNCTION or CREATE PROCEDURE statement directly to define a UDR. You must instead do this:

1. Create a source file with the CREATE FUNCTION or CREATE PROCEDURE statement.
2. Execute the CREATE ROUTINE FROM statement from an ESQL/C program to send the contents of this source file to the database server for execution. The file that you specify can contain only one CREATE FUNCTION or CREATE PROCEDURE statement.

The file specification that you provide is relative. If you include no pathname, the client application looks for the file in the current directory.

If you do not know at compile time whether the UDR in the file is a function or a procedure, use the CREATE ROUTINE FROM statement in the program. If you know whether the UDR is a function or a procedure, you can improve the readability of your code by using the matching SQL statement to access the source file:

- To access user-defined functions, use CREATE FUNCTION FROM.
- To access user-defined procedures, use CREATE PROCEDURE FROM.

When the IFX_EXTEND_ROLE configuration parameter is set to `1` or to `ON`, only users to whom the Database Server Administrator (DBSA) has granted the built-in EXTEND role can create external routines. In addition, you must hold at least the Resource access privilege on the database in which the routine will be registered. You must also hold the Usage privilege on the programming language in which the routine is written. (For the syntax of granting Usage privileges on the C language to a user or to a role, see Language-Level Privileges on page 549.)

Routines use the collating order that was in effect when they were created. See SET COLLATION statement on page 775 for information about using non-default collation.

Examples

The following statement registers at UDR by referencing the text in the `del_ord.sql` file.

```
EXEC SQL CREATE ROUTINE FROM 'del_ord.sql';
```

ESQL/C source code example:

```
#include <stdio.h>

main()
{

    printf( "CREATE ROUTINE FROM ESQL Program running.\n\n");
    EXEC SQL WHENEVER ERROR STOP;
    EXEC SQL connect to 'stores_demo';

    EXEC SQL CREATE ROUTINE FROM 'del_ord.sql';

    EXEC SQL disconnect current;
    printf("\nCREATE ROUTINE Sample Program over.\n\n");
```

```
   exit(0);
}

del_ord.sql

CREATE FUNCTION delete_order( p_order_num int) RETURNING int, int;
   DEFINE item_count int;
   SELECT count(*) INTO item_count FROM items
      WHERE order_num = p_order_num;
   DELETE FROM orders WHERE order_num = p_order_num;
   RETURN p_order_num, item_count;
END FUNCTION;
```

CREATE ROW TYPE statement

Use the CREATE ROW TYPE statement to create a named ROW type.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

CREATE ROW TYPE [IF NOT EXISTS] *[owner .] row_type* { <Named ROW type with no parent> <Named child ROW type> }

Named ROW type with no parent

" (<Field Definition>) "

Named child ROW type

" [(<Field Definition>)] **UNDER***supertype* "

Field Definition

" *field* "

" *data_type* [**NOT NULL**] "

Element	Description	Restrictions	Syntax
data_type	Data type of the *field*	See Restrictions on Serial and Simple-Large-Object Data Types on page 290.	Identifier on page 1231
field	Name of a field in *row_type*	Must be unique among field names of this ROW type and of its supertype	Identifier on page 1231

Element	Description	Restrictions	Syntax
owner	Authorization identifier of the owner of this ROW type	In an ANSI-compliant database, the combination *owner.row_type* must be unique among data-type objects	Owner name on page 1262
row_type	Name declared here for a new named ROW data type	See Procedure for Creating a Subtype on page 290. Must be unique among data type names in the database.	Identifier on page 1231
supertype	Name of the supertype of *row_type* within a data-type inheritance hierarchy	To create a child ROW type, this must exist in the database as a named ROW type, all of whose fields this *row_type* inherits	Data Type on page 985

Usage

You must hold the Resource privilege to use this statement. If the UNDER clause declares the new ROW type as a subtype of an existing named ROW type, you must also hold the UNDER privilege for that named ROW type. You are, by default, the owner of the named ROW types that you create, but a DBA who issues the CREATE ROW TYPE statement can designate another user as *owner*.

The CREATE ROW TYPE statement declares a named ROW data type and registers it in the **sysxtdtypes** system catalog table. You can assign a named ROW data type to a table or view to create a *typed table* or *typed view*. You can also define a column as a named ROW type.

The following example creates a named ROW type called **people_t** with four fields:

```
CREATE ROW TYPE people_t
(
   name     VARCHAR(40) NOT NULL,
   address  VARCHAR(35),
   city     VARCHAR(25),
   bdate    DATE
);
```

In some SQL code examples in this document, the SQL identifiers of named ROW types have `_t` as the last two characters. This suffix is only a documentation convention, and not a requirement of the database server.

Although you can assign a ROW type to a table to define the schema of the table, ROW data types are not the same as table rows. Table rows consist of one or more *columns*; ROW data types consist of one or more *fields*, defined using the Field Definition syntax.

A named ROW data type is valid in most contexts where you can specify a data type.

The following CREATE TABLE statement defines a typed table whose only column is of the **people_t** ROW type:

```
CREATE TABLE birthdays OF TYPE people_t
   LOCK MODE ROW;
```

Named ROW types are said to be *strongly typed*. No two named ROW types are equivalent, even if they are structurally equivalent. To compare values of a named ROW type with values of another named ROW type, or with values of an unnamed ROW type, you must use an explicit cast, so that both rows are of the same data type.

ROW types without identifiers are called *unnamed ROW types*. Any two unnamed ROW types are considered equivalent if they are structurally equivalent. For more information, see ROW Data Types on page 999.

Discretionary access privileges on the fields of named ROW types are the same as privileges on columns. For more information, see Table-Level Privileges on page 540. (To see what privileges you have on a column, check the **syscolauth** system catalog table, which is described in the *HCL® Informix® Guide to SQL: Reference*.)

If you include the optional `IF NOT EXISTS` keywords, the database server takes no action, rather than sending an exception to the application, if a named ROW of the specified name is already registered in the current database. In this case, the CREATE ROW TYPE statement has no effect, and does not register a new named ROW type in the **sysxtdtypes** system catalog table.

If no field is of the SERIAL, BIGSERIAL, or SERIAL8 data type, named ROW data types that the CREATE ROW TYPE statement defines can be the source type for DISTINCT ROW data types that the CREATE DISTINCT TYPE statement can define.

Important:

You cannot use the CREATE ROW TYPE statement to create an unnamed ROW type. Unnamed ROW types are created in the CREATE TABLE statement or in the ALTER TABLE statement by including the ROW constructor syntax segment in the definition of a column. This can be an unnamed ROW type column that you declare in the CREATE TABLE statement, or an unnamed ROW type that you add to the schema of an existing table by using the ALTER TABLE statement.

Field definition

When you define a new named ROW type with no UNDER clause, the Field Definition clause defines an ordered list of the data types and NOT NULL constraints of one or more fields in the new named ROW type.

If the UNDER clause identifies a *supertype*, however, the Field Definition clause can append one or more fields to the ordered list of fields that the new child ROW type inherits from its parent supertype within a named ROW type hierarchy.

If you omit the Field definition, the new subtype has only the fields that it inherits from the parent supertype that the UNDER clause specifies.

The NOT NULL constraint on the named ROW type field applies to the corresponding columns when a typed table of the named ROW type is created. If you omit the `NOT NULL` keywords, then by default, the field accepts NULL values.

The NOT NULL constraint is the only constraint that the CREATE ROW TYPE statement can define for a field of a named ROW type. To define any other constraints on the fields of a column of a named ROW type, you must use the CREATE TABLE statement, or in the ALTER TABLE statement.

The Informix® implementation of the SQL language supports no ALTER ROW TYPE statement for changing the definition of an existing named ROW type. For named ROW types that are not currently instantiated by any existing table, column, view, or named ROW type hierarchy in the database, you can use the DROP ROW TYPE statement to remove an existing named ROW

type from the system catalog. You can then use the CREATE ROW TYPE statement to define a replacement named ROW type.

See, however, the DROP ROW TYPE statement on page 483 for restrictions on dropping an existing named ROW type.

Privileges on named ROW data types

The discretionary access privileges required for operations on a typed table (a table that is assigned a named ROW data type) are the same as privileges on any table. For more information, see Table-Level Privileges on page 540. The following table shows which access privileges you need to create a named ROW type.

Task	Privileges Required
Create a named ROW type	Resource privilege on the database
Create a named ROW type as a subtype under a supertype	Under privilege on the supertype, as well as the Resource privilege

For information about Resource and Under privileges and the ALL keyword in the context of privileges, see the GRANT statement on page 537.

o find out what privileges exist on a ROW type, check the **sysxtdtypes** system catalog table for the *owner* name and the **sysxtdtypeauth** system catalog table for privileges on the ROW type that might have been granted to users or to roles.

To find out what privileges you have on a given table, check the **systabauth** system catalog table. For more information on system catalog tables, see the *HCL® Informix® Guide to SQL: Reference*.

Inheritance and Named ROW Types

A named ROW type can belong to an inheritance hierarchy, as either a subtype or a supertype. Use the UNDER clause in the CREATE ROW TYPE statement to create a named ROW data type as a subtype of an existing ROW data type.

The supertype must also be a named ROW data type. If you create a named ROW data type under an existing supertype, then the new type name *row_type* becomes the name of the subtype.

When you create a named ROW type as a subtype, the subtype inherits all fields of the supertype. In addition, you can add new fields to the subtype when you create it. The new fields are specific to the subtype alone.

You cannot substitute a ROW type in an inheritance hierarchy for its supertype or for its subtype. For example, consider a type hierarchy in which **person_t** is the supertype and **employee_t** is the subtype. If a column is of type **person_t**, the column can only contain **person_t** data. It cannot contain **employee_t** data. Likewise, if a column is of type **employee_t**, the column can only contain **employee_t** data. It cannot contain **person_t** data.

Creating a Subtype

In most cases, you add new fields when you create a named ROW type as a subtype of another named ROW type (its supertype). To create the fields of a named ROW type, use the field definition clause, as described in CREATE ROW TYPE

statement on page 286. When you create a subtype, you must use the UNDER keyword to associate the supertype with the named ROW type that you want to create. The next example creates the **employee_t** type under the **person_t** type:

```
CREATE ROW TYPE employee_t (salary NUMERIC(10,2),
   bonus NUMERIC(10,2)) UNDER person_t;
```

The **employee_t** type inherits all the fields of **person_t** and has two additional fields: **salary** and **bonus**; but the **person_t** type is not altered.

Type Hierarchies

When you create a subtype, you create a *type hierarchy*. In a type hierarchy, each subtype that you create inherits its properties from a single supertype. If you create a named ROW type **customer_t** under **person_t**, **customer_t** inherits all the fields of **person_t**. If you create another named ROW type, **salesrep_t** under **customer_t**, **salesrep_t** inherits all the fields of **customer_t**.

Thus, **salesrep_t** inherits all the fields that **customer_t** inherited from **person_t** as well as all the fields defined specifically for **customer_t**. For a discussion of type inheritance, refer to the *HCL® Informix® Guide to SQL: Tutorial*.

Procedure for Creating a Subtype

Before you create a named ROW type as a subtype in an inheritance hierarchy, check the following information:

- Verify that you are authorized to create new data types. You must have the Resource privilege on the database. You can find this information in the **sysusers** system catalog table.
- Verify that the supertype exists. You can find this information in the **sysxtdtypes** system catalog table.
- Verify that you are authorized to create subtypes to that supertype. You must have the Under privilege on the supertype. You can find this information in the **sysusers** system catalog table.
- Verify that the name that you declare for the named ROW type is unique. In an ANSI-compliant database, the *owner*.*type* combination must be unique within the database. In a database that is not ANSI-compliant, the name must be unique among data type names in the database. To verify whether the name for a new data type is unique, check the **sysxtdtypes** system catalog table. The name must not be the name of an existing data type.
- If you are defining fields for the ROW type, check that no duplicate field names exist in both new and inherited fields.

Important: When you create a subtype, you cannot redefine fields that it inherited for its supertype. If you attempt to redefine these fields, the database server returns an error.

You cannot apply constraints to named ROW data types, but you can specify constraints when you create or alter a table that uses the named ROW types. You can also specify NOT NULL constraints on individual fields of a ROW type.

Restrictions on Serial and Simple-Large-Object Data Types

Serial and simple-large-object data types cannot be nested within a table. Therefore, if a ROW type contains a BYTE, TEXT, SERIAL, BIGSERIAL, or SERIAL8 field, you cannot use the ROW type to define a column in a table that is not based on a ROW type. For example, the following code example produces an error:

```
CREATE ROW TYPE serialtype (s serial, s8 serial8);
CREATE TABLE tab1 (col1 serialtype); --INVALID CODE
```

You cannot create a ROW type that has a BYTE or TEXT value that is stored in a separate storage space. That is, you cannot use the IN clause to specify the storage location. For example, the following example produces an error:

```
CREATE ROW TYPE row1 (field1 byte IN blobspace1); --INVALID CODE
```

A table hierarchy can include no more than one SERIAL, BIGSERIAL, or SERIAL8 column. If a supertable has a SERIAL column, none of its subtables can contain a SERIAL column (but a subtable can have a BIGSERIAL or SERIAL8 column if no other subtable contains a BIGSERIAL or SERIAL8 column, respectively). Consequently, when you create the named ROW types on which the table hierarchy is to be based, they can contain at most one SERIAL and one BIGSERIAL or SERIAL8 field among them.

You cannot set the starting SERIAL, BIGSERIAL, or SERIAL8 value in the CREATE ROW TYPE statement. To modify the value for a serial field, you must use either the MODIFY clause of the ALTER TABLE statement, or else use the INSERT statement to insert a value that is larger than the current maximum (or default) serial value.

Serial fields in ROW types have performance implications across a table hierarchy. To insert data into a subtable whose supertable (or its supertable) contains the serial counter, the database server must also open the supertable, update the serial value, and close the supertable, thus adding extra overhead.

In contexts where these restrictions or performance issues for SERIAL, BIGSERIAL, or SERIAL8 data types conflict with your design goals, you might consider using sequence objects to emulate the functionality of serial fields or serial columns.

CREATE SCHEMA statement

Use the CREATE SCHEMA statement to issue a block of data definition language (DDL) and GRANT statements as a unit.

Use this statement with .

Syntax

(explicit id unique_418_Connect_42_crsc002) unique_418_Connect_42_crsc002(explicit id unique_418_Connect_42_crsc003)
unique_418_Connect_42_crsc003(explicit id unique_418_Connect_42_crsc004) unique_418_Connect_42_crsc004(explicit id unique_418_Connect_42_crsc006)
unique_418_Connect_42_crsc006(explicit id unique_418_Connect_42_crsc007) unique_418_Connect_42_crsc007(explicit id unique_418_Connect_42_crsc008)
unique_418_Connect_42_crsc008(explicit id unique_418_Connect_42_crsc009) unique_418_Connect_42_crsc009(explicit id unique_418_Connect_42_crsc011)
unique_418_Connect_42_crsc011(explicit id unique_418_Connect_42_crsc012) unique_418_Connect_42_crsc012(explicit id unique_418_Connect_42_crsc013)
unique_418_Connect_42_crsc013(explicit id unique_418_Connect_42_crsc014) unique_418_Connect_42_crsc014(explicit id unique_418_Connect_42_crsc015)
unique_418_Connect_42_crsc015 **CREATE SCHEMA AUTHORIZATION** *user* { | <CREATE TABLE Statement> [] | <CREATE VIEW Statement> []
| <GRANT Statement> [] | { [] | <CREATE INDEX Statement> [] | <CREATE SYNONYM Statement> [] | <CREATE TRIGGER Statement> [] | {
<CREATE SEQUENCE Statement> [] | <CREATE ROW TYPE Statement> [] | <CREATE OPAQUE TYPE Statement> [] | <CREATE DISTINCT TYPE
Statement> [] | <CREATE CAST Statement> [] } } } [;]

Element	Description	Restrictions	Syntax
user	User who owns the database objects that this statement creates	If you have DBA privileges, you can specify the name of any user. Otherwise, you must have the Resource privilege, and you must specify your own user name.	Owner name on page 1262

Usage

The CREATE SCHEMA statement allows the DBA to specify an owner for all database objects that the CREATE SCHEMA statement creates. You cannot issue CREATE SCHEMA until you have created the database that stores the objects.

Users with the Resource privilege can create a schema for themselves. In this case, *user* must be the name of the person with the Resource privilege who is running the CREATE SCHEMA statement. Anyone with the DBA privilege can also create a schema for someone else. In this case, *user* can specify a user other than the person who is running the CREATE SCHEMA statement.

You can put CREATE and GRANT statements in any logical order, as the following example shows. Statements are considered part of the CREATE SCHEMA statement until a semicolon (;) or an end-of-file symbol is reached.

```
CREATE SCHEMA AUTHORIZATION sarah
   CREATE TABLE mytable (mytime DATE, mytext TEXT)
   GRANT SELECT, UPDATE, DELETE ON mytable TO rick
   CREATE VIEW myview AS
      SELECT * FROM mytable WHERE mytime > '12/31/2004'
   CREATE INDEX idxtime ON mytable (mytime);
```

Creating Database Objects Within CREATE SCHEMA

All database objects that a CREATE SCHEMA statement creates are owned by *user*, even if you do not explicitly name each database object. If you are the DBA, you can create database objects for another user. If you are not the DBA, specifying an owner other than yourself results in an error message.

Suppose that user **oswald** holds the DBA access privilege and issues the following CREATE SCHEMA statement:

```
CREATE SCHEMA AUTHORIZATION hilda
   CREATE ROW TYPE IF NOT EXISTS
      hildago (field1 INT,
               field2 CHAR(18) NOT NULL,
               field3 DATE);
```

The owner of this named ROW type is the user whose authorization identifier is **hilda**, rather than user **oswald** who issued the CREATE ROW TYPE statement. This example would fail if user **oswald** does not hold the DBA privilege.

You can only grant access privileges with the CREATE SCHEMA statement; you cannot use CREATE SCHEMA to revoke or to drop access privileges.

If you create a database object or use the GRANT statement outside a CREATE SCHEMA statement, you receive warnings if you use the **-ansi** flag or set **DBANSIWARN**.

CREATE SECURITY LABEL statement

Use the CREATE SECURITY LABEL statement to define a new security label for a specified security policy in the current database and to identify its components and the elements of its components.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
CREATE SECURITY LABEL [IF NOT EXISTS] policy. label COMPONENTcomponent element
```

Element	Description	Restrictions	Syntax
component	A security label component	Must already exist in the database as a component of the specified *policy*, and be unique among names of components of this *label*.	Identifier on page 1231
label	Name you declare here for this label	Must be unique among security label names for this security *policy*	Identifier on page 1231
element	An element of the specified *component*	Must have been defined when its *component* was defined or was last altered. If *component* is an array, only a single *element* can be specified.	Quoted String on page 1202
policy	The security policy of this *label*	Must already exist in the database	Identifier on page 1231

Usage

A *security label* is a named database object that supports a specified security policy. A security label can be applied to a user, or to a row or to a column (or to both a row and a column) of a table in the database. When a user who holds a security label attempts to access data that has a security label, the database server takes into account the security label of the column or row and the security label of the user in determining whether to allow the user to access the data.

Every security label stores the following categories of information:

- It identifies an existing security policy that the label supports.
- It identifies at least one, but no more than 16 existing components of the security policy that the label supports.
- It identifies one or more existing elements of each component of the security label. (Only security label components of type SET or TREE can include more than one element in the same security label.)

Only DBSECADM can issue this statement. When the CREATE SECURITY LABEL statement executes successfully, it registers the specified *label* name, the numeric identifier of the associated security *policy*, and the cardinality of its security label *components* in the **sysseclabels** system catalog table.

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if a security label of the specified name is already registered in the current database.

Components and Elements of a Security Label

Like a security policy, a security label must have at least one component, but no more than 16. The CREATE SECURITY LABEL statement cannot list security label components that are not components of the specified security policy. The same *component* name cannot be specified more than once in the same CREATE SECURITY LABEL statement. These components must already exist in the database, where DBSECADM can register them with the CREATE SECURITY LABEL COMPONENT statement.

Security label components can be of type ARRAY, SET, or TREE, as described in CREATE SECURITY LABEL COMPONENT. For a *component* of type ARRAY, the *element* list can identify only a single element. For components of type SET or TREE, the *element* list can identify multiple component elements that were defined when the component was created (or when it was last altered). See the CREATE SECURITY LABEL COMPONENT statement for more information about the structure and semantics of security label components.

The following example creates a security label called **label1** for a security policy called **MegaCorp**. The label uses two security label components, called **levels** and **compartments**, each with one element, called **VP** and **Marketing** respectively:

```
CREATE SECURITY LABEL MegaCorp.label1
   COMPONENT levels 'VP',
   COMPONENT compartments 'Marketing';
```

For this example to be valid, the **levels** and **compartments** components, and their security label components, **VP** and **Marketing** elements, must have been defined in previously executed CREATE SECURITY LABEL COMPONENT statements.

In the next example, DBSECADM creates a security label called **label2** for the same **MegaCorp** security policy. This label uses three security label components, called **levels**, **compartments**, and **groups**, where two of these components have one element, and another has two:

```
CREATE SECURITY LABEL MegaCorp.label2
   COMPONENT level 'Director',
   COMPONENT compartments 'HR', 'Finance',
   COMPONENT groups 'EntireRegion';
```

These examples illustrate that the components of a security label can be a subset of the components of the security policy that the label supports, and that more than one security label can support the same security policy.

CREATE SECURITY LABEL COMPONENT statement

Use the CREATE SECURITY LABEL COMPONENT statement to define a new security label component in the current database and to define the elements the component.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
CREATE SECURITY LABEL COMPONENT [IF NOT EXISTS]  component { ARRAY[ element ] | SET{ element } | TREE( elementROOT
, elementUNDERelement ) }
```

Element	Description	Restrictions	Syntax
component	Name declared here for this component	Must be unique among the names of security label components in the database.	Identifier on page 1231
element	Component element that is defined here	Must be unique among elements of this *component*, and no longer than 32 bytes. The left (`(`) and right (`)`) parentheses, comma (`,`), and colon (`:`) symbols are not valid characters.	Quoted String on page 1202

Usage

Only the DBSECADM can issue the CREATE SECURITY LABEL COMPONENT statement, which defines a *security label component*. This is a database object that defines one or more logical categories whose values can be used in a security policy to determine whether a user's request to read or write data is accepted or rejected. The set of all the valid individual values that the security component can have is defined by the set of *security label elements* that this statement specifies for the component.

The logical categories that security label components implement are identified by DBSECADM in the process of designing a *security policy*, which is the core construct of label-based access control (LBAC). To implement this security feature in the database, however, DBSECADM must create security objects in the following sequence:

1. A set of one or more *security components*, each of which can be defined by the CREATE SECURITY LABEL COMPONENT statement. This statement specifies the name of a security component, the structure of its range of values, and the possible values for this component that can be assigned to a security label that applies a security policy to data or to a user.
2. One or more *security policies*, each of which can be defined by the CREATE SECURITY POLICY statement, which specifies a list of one or more components and a set of rules that the security policy applies to data and to users who attempt read or write operations on data that the security policy protects in the database. A security policy always includes all the elements of a component that CREATE SECURITY POLICY specifies.
3. A *security label* can be defined by the CREATE SECURITY LABEL statement, which specifies one or more values for each of one or more components of the security policy that the label supports. The security label can be applied to data and to users. All the components of a security label must be components of the same security policy, but multiple security policies and multiple security labels can share the same component. A security label typically includes only a subset of the elements of a security component that CREATE SECURITY LABEL COMPONENT defines.

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if a security label component of the specified name is already registered in the current database.

See the GRANT Security and REVOKE Security statements for information on how security labels and exemptions from the rules of a security policy define the LBAC credentials of a user or of a role.

See the CREATE TABLE and ALTER TABLE statements for information on how security labels can be associated with a database table or with an individual data row in a table.

Types and Elements of Security Label Components

A security label component itself consists of one or more *elements* that the CREATE SECURITY LABEL COMPONENT statement declares as string constants. These elements define the set of values that are valid for the component,

When the CREATE SECURITY LABEL statement executes successfully, Informix® updates the system catalog of the database with the following new entries:

- It creates a new row in the **sysseclabelcomponents** table to register the new component.
- For each *element* of the new component, it creates a new row in the **sysseclabelcomponentelements** table.

The security label component must be defined as one of the three component types. The ARRAY, SET, or TREE keyword that immediately follows the declaration of the *component* name specifies the component type, which must be followed by a list of the *elements* of the security component. These elements define the set of values that the component can have within a security policy. For all three types of security label components, the set of elements is under the following restrictions:

- The security component can have no more than 64 elements.
- Each element of a security component is a quoted string constant of no more than 32 bytes.
- Characters in the quoted string constant cannot include the left (`(`) or right (`)`) parentheses, comma (`,`), or colon (`:`) symbols, but other symbols that the **DB_LOCALE** setting supports are valid, including the blank space (ASCII 32) character.
- Each element must be unique among elements of the same security label component, but the same quoted string constant value can also be an element of other security label components.

The definition of each element within the component implies a level of data sensitivity that a security label associates with a database table or with an individual data row, and also affects the security credentials of users who hold a security label to read or write data that is protected by the same label or by a different label that specifies one or more elements of the component.

Like other database Data Definition Language statements of SQL that can define database objects, CREATE SECURITY LABEL COMPONENT must specify a literal value for each component element, rather than a placeholder. To change the definition of an existing security label component, DBSECADM can use the ALTER SECURITY LABEL COMPONENT to insert a new element into an ARRAY, SET, or TREE component. To drop or rename one or more individual elements of a component, however, DBSECADM must use the DROP SECURITY LABEL COMPONENT statement to destroy the existing component, and then reissue the CREATE SECURITY LABEL COMPONENT statement to create a new component that defines the required set of element values within an ARRAY, SET, or TREE component structure.

ARRAY Components

A security label component of type ARRAY is an ordered set of no more than 64 elements. Each element defines a value that is valid for that component within a security policy. The order in which elements are declared is significant, because it defines a descending order of data sensitivity, with each successive element ranking lower in data sensitivity than the preceding element. The set of label elements of the array and their comma separators must be enclosed between a pair of bracket (`[ ... ]`) symbols.

When an ARRAY component is specified in the definition of a security label, the label can specify no more than one element of that component as the value of the component.

The following example defines a security label component of type ARRAY called **aquilae** that is an ordered set of five elements called **imperator**, **tribunus**, **centurio**, **miles**, and **asinus**:

```
CREATE SECURITY LABEL COMPONENT aquilae
   ARRAY [ "imperator", "tribunus", "centurio", "miles", "asinus" ];
```

Here the component element with the highest data sensitivity is **imperator** and **asinus** has the lowest data sensitivity, with the data sensitivity of **tribunus** ranking above that of **centurio** but below that of **imperator**.

A component of type ARRAY can be appropriate in contexts where some dimension of a multidimensional security policy can be mapped onto a single scale that is monotonically descending.

SET Components

A security label component of type SET is an unordered set of no more than 64 elements. Each element of the SET is a string constant of no more than 32 bytes, and must be unique within the component, but the same value can be used in other components. The order in which the elements of a SET component are declared is not significant in regard to the data sensitivity of the categories that these elements identify. The elements and their comma separators must be enclosed between a pair of braces (`{ ... }`) symbols.

When a SET component is specified in the definition of a security label, the label can specify one or multiple elements of that component as valid values for the component.

In the following example, DBSECADM defines a security label component called **departments** that is an unordered set of three elements, called **Marketing**, **HR**, and **finance**:

```
CREATE SECURITY LABEL COMPONENT departments
   SET { 'Marketing', 'HR', 'Finance' };
```

Like all components of type SET, the order in which these elements are declared implies no relative rank in data sensitivity.

A component of type SET can be appropriate in contexts where some dimension of a multidimensional security policy can be represented as nominal categories, without any logical basis for ordering them on a monotonic scale, nor for arranging them in a hierarchy.

TREE Components

A security label component of type TREE has the logical topology of a hierarchy (that is, a simple graph with no loops) that has a single root node and no more than 63 additional nodes. The string constant for the root node must be listed first and must be followed by the ROOT keyword. The string constant for each subsequently declared node must be followed by the keyword UNDER and by the string constant for some previously declared node. The set of elements of the TREE component, including their ROOT and UNDER keywords and comma separators, must be enclosed between a pair of parenthesis (`( ... )`) symbols.

The label element specified after the UNDER keyword is called the *parent* of the label element that precedes the same UNDER keyword (which is called the *child* of that parent element). The CREATE SECURITY LABEL COMPONENT statement fails with an error if a node name that follows the UNDER keyword has not already been declared in the same statement.

The string constant that designates the root node of a tree component has the highest data sensitivity. For a user to read or write protected data, each tree component of the user security label must include at least one of the elements in the tree component of the data row security label, or the ancestor of one such element. For example, if "**Beta**" is declared UNDER "**Alpha**" and "**Gamma**" is declared UNDER "**Beta**" then "**Gamma**" also ranks below "**Alpha**" in data sensitivity. Only elements that are in the same chain of parent-child relationships can be compared in their data sensitivity.

The next example defines a security label component called **Oakland** as a tree structure with six nodes:

```
CREATE SECURITY LABEL COMPONENT Oakland
TREE ( 'Port' ROOT,
          'Downtown' UNDER 'Port',
          'Airport' UNDER 'Port',
          'Estuary' UNDER 'Airport',
          'Avenues' UNDER 'Downtown',
          'Hills' UNDER 'Avenues');
```

Here the root node is **Port**, which has the highest data sensitivity. Within this hierarchy, the **Downtown**, **Avenues**, and **Hills** elements represent descending levels of data sensitivity, and the **Airport** element has a higher data sensitivity than the **Estuary** element. In this example, the four component elements that the UNDER keyword designates as parent nodes are each declared before being included in UNDER specifications. A modified version of this example would also be valid if the **Avenues** node declaration preceded the **Airport** node declaration, but an error would result if the **Hills** node declaration had preceded the **Avenues** node declaration.

A component of type TREE can be appropriate in contexts where some dimension of a multidimensional security policy can be mapped to a single logical hierarchy, or to a group of hierarchies that share a common root.

CREATE SECURITY POLICY statement

Use the CREATE SECURITY POLICY statement to define a new security policy in the current database and to identify its security label components and access rules.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
CREATE SECURITY POLICY [IF NOT EXISTS] policy COMPONENTS component (explicit id) WITH IDSLBACRULES { RESTRICT NOT AUTHORIZED
WRITE SECURITY LABEL | [OVERRIDE NOT AUTHORIZED WRITE SECURITY LABEL] }
```

Element	Description	Restrictions	Syntax
component	A security label component	Must already exist in the database, and be unique among the names of components for this *policy*	Identifier on page 1231
policy	Name declared here for a security policy	Must be unique among the names of security policies in the database	Identifier on page 1231

Usage

A *security policy* is a named database object that stores the following information:

- It defines a set of security label components that comprise a security label.
- It associates that security label with a set of access rules.

For tables that are protected by a security policy, the access rules enable Informix® to compare the security credentials of a user with the security label of a row or column. The security policy is applied to determine whether a user who holds a given security label can read or write data in a row or column that is labeled with a security label. A security policy has no effect on data that has no security label.

No more than one security policy can be attached to a table at any point in time, and a security policy can include no more than 16 security label components.

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if a security policy of the specified name is already registered in the current database. In this case, no new security policy is created is created, and the CREATE SECURITY POLICY statement has no effect on the existing security policy that it referenced.

Only DBSECADM can issue this statement. When the CREATE SECURITY POLICY statement executes successfully, Informix® makes the following updates to the system catalog of the current database:

- Registers the specified *policy* name and the cardinality of its security label components in the **syssecpolicies** table
- Creates for each *component* a new row in the **syssecpolicycomponentrules** table.

Example of creating a new security policy

This is the sequence in which LBAC security objects associated with a specific security label must be created:

- Security label components
- Security policy
- Security label

For example, the following CREATE SECURITY LABEL COMPONENT statement registers a security label component named **Departments** of type `SET` in the **sysseclabelcomponents** system catalog table of the database:

```
CREATE SECURITY LABEL COMPONENT departments
   SET { 'Sales','Legal','IT','CanineResources' };
```

The same statement also registers the four elements of the **departments** component, namely **'Sales'**, **'Legal'**, **'IT'**, and **'CanineResources'** in the **sysseclabelcomponentelements** system catalog table.

Note that this example of a security label component definition specifies no security policy, because only the CREATE SECURITY POLICY statement can associate a component with a security policy.

For more information about security label components, see Types and Elements of Security Label Components on page 296 and CREATE SECURITY LABEL COMPONENT statement on page 294.

The following SQL statement registers a new security policy called **WatchDog** in the current database:

```
CREATE SECURITY POLICY WatchDog
   COMPONENTS departments
      WITH IDSLBACRULES;
```

Here the security label component **departments** must already exist in the database, and the name **WatchDog** must be unique among the identifiers of existing security policies in the database. For an explanation of the `WITH IDSLBACRULES` keywords, see Rules Associated with a Security Policy on page 301, which also provides a more general example of defining a security policy with multiple components.

Note that the definition of the **WatchDog** security policy includes no explicit references to security labels. Any security labels that the CREATE SECURITY LABEL statement associates with the **WatchDog** security policy must reference **departments** as their *component*, because this security policy has no other component. The following example illustrates this requirement for **WatchDog** labels:

```
CREATE SECURITY LABEL WatchDog.label9
   COMPONENT departments 'Sales','CanineResources';
```

This statement declares **label9** as the name of a new label of the **WatchDog** security policy. It also registers the following information in the **sysseclabels** system catalog table:

- this label name,
- and the numeric identifier of the associated **WatchDog** security policy,
- and the cardinality of its two (2) security label components.

The `COMPONENT` clause in the same example specifies that the **label9** label is valid for both the **'Sales'** and for the **'CanineResources'** elements of the **departments** component of the **WatchDog** security policy.

For more information about security labels, see Components and Elements of a Security Label on page 294 and CREATE SECURITY LABEL statement on page 293.

For information on protecting a new table by attaching a security policy when the table is being created, see the SECURITY POLICY Clause on page 349 of the CREATE TABLE statement.

For the syntax to attach or to drop a security policy for an existing table, see SECURITY POLICY Clause on page 154 of the ALTER TABLE statement.

Security Label Components of a Security Policy

The CREATE SECURITY POLICY statement must specify at least one (but no more than 16) security label components. These components must already exist in the database, where DBSECADM can register them with the CREATE SECURITY LABEL COMPONENT statement. The same *component* name cannot be specified more than once in the same CREATE SECURITY POLICY statement.

See the section CREATE SECURITY LABEL COMPONENT for more information about the structure and semantics of security label components.

Rules Associated with a Security Policy

The WITH IDSLBACRULES keywords specify the read access rules and write access rules that the new security policy enforces. If you do not specify them, these keywords are in effect by default, because the **IDSLBACRULES** access rules are the only access rules that the a security policy can support.

The following **IDSLBACRULES** access rules for read access, called **IDSLBACREAD**, apply when data values are read from labeled rows or columns in SELECT, UPDATE, or DELETE operations:

- **IDSLBACREADARRAY**: Each array component of the user security label must be greater than or equal to the array component of the data row security label. That is, only data at or below the level of the user can be read.
- **IDSLBACREADTREE**: Each tree component of the user security label must include at least one of the elements in the tree component of the data row security label (or the ancestor of one such element).
- **IDSLBACREADSET**: Each set component of the user security label must include the set component of the data row security label.

The following **IDSLBACRULES** access rules for write access, called **IDSLBACWRITE**, apply when data values are written to labeled rows or columns in INSERT, UPDATE, or DELETE operations:

- **IDSLBACWRITEARRAY**: Each array component of the user security label must be equal to the array component of the data row security label. That is, only data at the same level as the user can be written.
- **IDSLBACWRITETREE**: Each tree component of the user security label must include at least one of the elements in the tree component of the data row security label (or the ancestor of one such element).
- **IDSLBACWRITESET**: Each set component of the user security label must include the set component of the data row security label.

If DBSECADM omits the WITH IDSLBACRULES keywords, then those rules are in effect by default. If any specification except IDSLBACRULES follows the WITH keyword, however, the CREATE SECURITY POLICY statement fails with an error, and no security policy is created.

Besides the explicit or default WITH IDSLBACRULES keywords, the CREATE SECURITY POLICY statement must also specify the write access rule to enforce when a user is not authorized to write the explicitly specified security label provided in the DELETE, INSERT, or UPDATE statement for a table protected with this security policy. The security label of the user and the exemption credentials that the user holds determine whether the user has write access to an explicitly provided security label.

- If the CREATE SECURITY POLICY statement specifies OVERRIDE NOT AUTHORIZED WRITE SECURITY LABEL, then Informix® uses the value of the user security label, rather than the security label that is explicitly specified in the DELETE, INSERT, or UPDATE statement, to determine whether the user has write-access to data values that are protected by a security label in the DELETE, INSERT, or UPDATE operation.
- The default is RESTRICT NOT AUTHORIZED WRITE SECURITY LABEL. If you specify these keywords explicitly, or if they are in effect by default, then DELETE, INSERT, or UPDATE statements fail with an error if the user is not authorized to write data in a row or column that has the explicitly specified security label.

The following example creates a security policy called **MegaCorp** that uses three security label components, with no OVERRIDE provision for the user security label to provide write access in DELETE, INSERT, or UPDATE operations on data whose explicitly specified security label does not authorize write access for that user:

```
CREATE SECURITY POLICY MegaCorp
   COMPONENTS levels, compartments, groups
      WITH IDSLBACRULES;
```

For this example to be valid, the **levels**, **compartments**, and **groups** security label components (or components that have been renamed to these identifiers) must have been previously defined by the CREATE SECURITY LABEL COMPONENT statement.

CREATE SEQUENCE statement

Use the CREATE SEQUENCE statement to create a sequence database object from which multiple users can generate unique integers.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

CREATE SEQUENCE [IF NOT EXISTS] [*owner* .] *sequence* [{ | **INCREMENT BY** *step* | **START WITH** *origin* | { **MAXVALUE** *max* | **NOMAXVALUE** } | { **MINVALUE** *min* | **NOMINVALUE** } | { **NOCYCLE** | **CYCLE** } | { **CACHE** *size* | **NOCACHE** } | { **NOORDER** | **ORDER** } } (explicit id)]

Element	Description	Restrictions	Syntax
max	Upper limit of values	Must be an integer > *origin*	Literal Number on page 1198
min	Lower limit of values	Must be an integer less than *origin*	Literal Number on page 1198
origin	First number in the sequence	Must be an integer in INT8 or BIGINT range	Literal Number on page 1198
owner	Owner of *sequence*	Must be an authorization identifier	Owner name on page 1262
sequence	Name that you declare here for the new sequence	Must be unique among sequence, table, view, and synonym names	Identifier on page 1231
size	Number of values that are preallocated in memory	Integer > 1, but < cardinality of a cycle (= |(*max* - *min*)/*step*|)	Literal Number on page 1198
step	Interval between successive values	Nonzero integer in INT range	Literal Number on page 1198

Usage

A sequence (sometimes called a *sequence generator* or *sequence object*) returns a monotonically ascending or descending series of unique integers, one at a time. The CREATE SEQUENCE statement defines a new sequence object, declares its identifier, and registers it in the **syssequences** system catalog table.

Authorized users of a sequence can request a new value by including the *sequence***.NEXTVAL** expression in DML statements. The *sequence***.CURRVAL** expression returns the current value of the specified *sequence*. **NEXTVAL** and **CURRVAL** expressions are valid only within SELECT, DELETE, INSERT, and UPDATE statements; Informix® returns an error if you attempt to invoke the built-in **NEXTVAL** or **CURRVAL** functions in any other context.

Generated values logically resemble the BIGSERIAL or SERIAL8 data type, but can be negative, and are unique within the sequence. Because the database server generates the values, sequences support a much higher level of concurrency than a serial column can. The values are independent of transactions; a generated value cannot be rolled back, even if the transaction in which it was generated fails.

You can use a sequence to generate primary key values automatically, using one sequence for many tables, or each table can have its own sequence.

CREATE SEQUENCE can specify the following characteristics of a sequence:

- Initial value
- Size and sign of the increment between values
- Maximum and minimum values
- Whether the sequence recycles values after reaching its limit
- How many values are preallocated in memory for rapid access.

A database can support multiple sequences concurrently, but the name of a sequence (or in an ANSI-compliant database, the *owner***.***sequence* combination) must be unique within the current database among the names of tables, temporary tables, views, synonyms, and sequences.

An error occurs if you include contradictory options, such as specifying both the MINVALUE and NOMINVALUE options, or both CACHE and NOCACHE.

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if a sequence object of the specified name is already registered in the current database, or if the specified name is the identifier of a table, view, or synonym in the current database.

Example

The following example creates a sequence, inserts values from the sequence into the table, and selects all rows and columns from the table.

```
CREATE SEQUENCE seq_2
   INCREMENT BY 1 START WITH 1
   MAXVALUE 30 MINVALUE 0
   NOCYCLE CACHE 10 ORDER;

CREATE TABLE tab1 (col1 int, col2 int);
INSERT INTO tab1 VALUES (0, 0);

INSERT INTO tab1 (col1, col2) VALUES (seq_2.NEXTVAL, seq_2.NEXTVAL)

SELECT * FROM tab1;
```

```
      col1        col2

         0           0
         1           1
```

INCREMENT BY Option

Use the INCREMENT BY option to specify the interval between successive numbers in the sequence. The BY keyword is optional. The interval, or *step* value, can be a positive whole number (for an *ascending* sequence) or a negative whole number (for a *descending* sequence) in the INT8 range. If you do not specify any *step* value, the default interval between successive generated values is `1`, and the sequence is an ascending sequence.

START WITH Option

Use the START WITH option to specify the first number of the sequence. This *origin* value must be an integer within the INT8 range that is greater than or equal to the *min* value (for an ascending sequence) or that is less than or equal to the *max* value (for a descending sequence), if *min* or *max* is specified in the CREATE SEQUENCE statement. The WITH keyword is optional.

If you do not specify an *origin* value, the default initial value is *min* for an ascending sequence or *max* for a descending sequence. (The MAXVALUE or NOMAXVALUE Option on page 304 and MINVALUE or NOMINVALUE Option on page 304 sections that follow describe the *max* and *min* specifications respectively.)

MAXVALUE or NOMAXVALUE Option

Use the MAXVALUE option to specify the upper limit of values in a sequence. The maximum value, or *max*, must be an integer in the INT8 range that is greater than the value of the *origin*.

If you do not specify a *max* value, the default is NOMAXVALUE. This default setting supports values that are less than or equal to `2e64` for ascending sequences, or less than or equal to `-1` for descending sequences.

MINVALUE or NOMINVALUE Option

Use the MINVALUE option to specify the lower limit of values, or *min*. This integer must be in the INT8 range and be less than the value of *origin*.

If you do not specify a *min* value, the default is NOMINVALUE. This default setting supports values that are greater than or equal to `1` for ascending sequences, or greater than or equal to `-(2e64)` for descending sequences.

CYCLE or NOCYCLE Option

Use the CYCLE option to continue generating sequence values after the sequence reaches the maximum (ascending) or minimum (descending) limit. After an ascending sequence reaches the *max* value, it generates the *min* value for the next sequence value. After a descending sequence reaches the *min* value, it generates the *max* value for the next sequence value.

The default is NOCYCLE. At this default setting, the sequence cannot generate more values after reaching the declared limit. Once the sequence reaches the limit, the next reference to *sequence*.**NEXTVAL** returns an error.

CACHE or NOCACHE Option

Use the CACHE option to specify the number of new sequence values that are preallocated in memory for rapid access. This option can enhance the performance of a sequence that grows quickly.

The cache *size* must be a positive whole number in the INT range. If you specify the CYCLE option, then *size* must be less than the number of values in a cycle (or less than |(*max* - *min*)/*step*|). The minimum is `2` preallocated values. The default is `20` preallocated values.

The NOCACHE keyword specifies that no generated values (that is, zero) are preallocated in memory for this sequence object.

ORDER or NOORDER Option

These keywords have no effect on the behavior of the sequence. The sequence always issues values to users in the order of their requests, as if the ORDER keyword were always specified. The ORDER and NOORDER keywords are accepted by the CREATE SEQUENCE statement for compatibility with implementations of sequence objects in other dialects of SQL.

CREATE SYNONYM statement

Use the CREATE SYNONYM statement to declare and register an alternative name for an existing table, view, or sequence object.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_421_Connect_42_cresyn001) unique_421_Connect_42_cresyn001
```
CREATE { [ PUBLIC | [ PRIVATE } SYNONYM [IF NOT EXISTS]
synonym FOR { table | view | sequence }
```

Element	Description	Restrictions	Syntax
sequence	Name of a local sequence	Must exist in the current database	Identifier on page 1231
table, *view*	Name of database table, external table, or view for which *synonym* is being created	Must be registered in the current database, or in a database specified in a qualifier	Database Object Name on page 1225
synonym	Synonym declared here for the name of *table*, *view*, or *sequence*	Must be unique among table object names in the database; see also **Usage** notes.	Database Object Name on page 1225

Usage

Users have the same privileges for a synonym that they have for the database object that the synonym references. The **syssyntable** and **systables** system catalog tables maintain information about synonyms.

You cannot create a synonym for a synonym in the same database.

The identifier of the synonym must be unique among the names of tables, temporary tables, external tables, views, and sequence objects in the same database. (See, however, the section Synonyms with the Same Name on page 308.)

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if a synonym of the specified name is already registered in the current database, or if the specified name is the identifier of a table, view, or sequence object in the current database.

Once a synonym is created, it persists until the owner executes the DROP SYNONYM statement. (This persistence distinguishes a synonym from an alias that you can declare in the FROM clause of a SELECT statement. The alias is in scope only while that SELECT statement is executing.)

If a synonym refers to a table, view, or sequence in the same database, however, the synonym is automatically dropped if the referenced table, view, or sequence object is dropped. For additional information, see the section Synonyms for objects outside the current database on page 306.

Synonyms for objects outside the current database

A synonym can be created for any table or view in any database of the database server to which your session is currently connected.

This example declares a synonym for a table outside your current database, in the **payables** database of your current database server.

```
CREATE SYNONYM mysum FOR payables:jean.summary;
```

You can also create a synonym for an external table that the CREATE EXTERNAL TABLE statement registered in the current database. (The external table is registered in the system catalog of the database where it was created, but it is not stored in any database.)

You can also create a synonym for a table or view that exists in a database of a database server that is not your current database server. Both database servers must be online when you create the synonym. In a network, the remote database server verifies that the table or view referenced by the synonym exists when you create the synonym. The next example creates a synonym for a table in a database of a remote database server:

```
CREATE SYNONYM mysum FOR payables@phoenix:jean.summary;
```

The identifier **mysum** now refers to the table **jean.summary**, which is in the **payables** database on the **phoenix** database server. If the **summary** table is dropped from the **payables** database, the **mysum** synonym is left intact. Subsequent attempts to use **mysum** return the error: `Table not found`.

Informix®, however, does not support synonyms for these table objects :

- Typed tables (including any table that is part of a table hierarchy)
- Tables or views with columns of any extended data types
- Sequence objects outside the local database

PUBLIC and PRIVATE Synonyms

If you use the `PRIVATE` keyword to declare a synonym in a database that is not ANSI-compliant, the unqualified synonym can be used by its owner. Other users must qualify the synonym with the name of the owner.

If you use the `PUBLIC` keyword (or no keyword at all), anyone who has access to the database can use your synonym. If the database is not ANSI-compliant, a user does not need to know the name of the owner of a public synonym.

In an ANSI-compliant database, all synonyms are private. If you use the `PUBLIC` or `PRIVATE` keywords, the database server issues a syntax error.

Examples of PRIVATE and PUBLIC synonyms

Suppose that in a database that is not ANSI-compliant, user **primus** issues the following SQL statements:

```
CREATE SEQUENCE IF NOT EXISTS MySequence
   INCREMENT BY 1 START WITH 1
   MAXVALUE 8000 MINVALUE 0
   NOCYCLE CACHE 20 ORDER;

CREATE PRIVATE SYNONYM IF NOT EXISTS anaphora FOR MySequence;
```

Now user **primus** is the owner of two new objects in the current database:

- a sequence object called **MySequence**,
- and a private synonym called **anaphora** for the **MySequence** sequence.

Suppose that in the same database, user **primus** also issues the following SQL statements

- to create a permanent table called **twoSmall**,
- and to declare **litotes** as a private synonym for the **twoSmall** table:

```
CREATE TABLE IF NOT EXISTS twoSmall (c1 SMALLINT, c2 SMALLINT);
INSERT INTO twoSmall VALUES (0, 0);

CREATE PRIVATE SYNONYM litotes FOR twoSmall;
```

In the same database that is not ANSI-compliant, another user could reference both the **litotes** and **anaphora** synonyms in the following INSERT statement, where **twoSmall** is the target table:

```
INSERT INTO primus.litotes (col1, col2)
   VALUES (primus.anaphora.NEXTVAL, primus.anaphora.NEXTVAL);
```

Although both synonyms in the INSERT statement were created as `PRIVATE`, the user can access the objects that those synonyms reference by qualifying the synonym names with the authorization identifier of **primus**, who is the owner of those synonyms.

If instead of creating the private synonyms in the previous examples, user **primus** had implicitly or explicitly created both **anaphora** and **litotes** as `PUBLIC` synonyms, any other user who held sufficient access privileges could omit the authorization identifier of **primus** in the previous INSERT statement:

```
INSERT INTO litotes (col1, col2)
   VALUES (anaphora.NEXTVAL, anaphora.NEXTVAL);
```

In a database that was not created as MODE ANSI, the following example shows the syntax for user **primus** to create **synecdoche** as a `PUBLIC` synonym for the **MySequence** object:

```
CREATE SYNONYM IF NOT EXISTS synecdoche FOR MySequence;
```

In the example above, **synecdoche** is a public synonym by default, because neither the `PUBLIC` nor `PRIVATE` keyword was specified.

In the same database, the next statement creates **zeugma** as an explicit `PUBLIC` synonym:

```
CREATE PUBLIC SYNONYM IF NOT EXISTS zeugma FOR MySequence;
```

In the same database, the next statement creates **pleonasm** as a `PRIVATE` synonym for the same sequence object:

```
CREATE PRIVATE SYNONYM IF NOT EXISTS pleonasm FOR MySequence;
```

In a database created as MODE ANSI, the next statement creates **asyndeton** as a `PRIVATE` synonym, because ANSI-compliant databases do not support `PUBLIC` synonyms:

```
CREATE SYNONYM asyndeton FOR MySequence;
```

In a database created as MODE ANSI, the next statement fails with a syntax error, because the CREATE SYNONYM statement does not support the `PRIVATE` or `PUBLIC` keywords in ANSI-compliant databases:

```
CREATE PRIVATE SYNONYM litotes FOR MySequence;
```

For information about the scope of reference of the identifiers of public and private synonyms in databases that are not ANSI-compliant, see the topic Synonyms with the Same Name on page 308.

Synonyms with the Same Name

In an ANSI-compliant database, the *owner*.*synonym* combination must be unique among all synonyms, tables, views, and sequences. You must specify *owner* when you refer to a synonym that you do not own, as in this example:

```
CREATE SYNONYM emp FOR accting.employee
```

In a database that is not ANSI-compliant, no two public synonyms can have the same identifier, and the identifier of a synonym must also be unique among the names of tables, views, and sequences in the same database.

The *owner*.*synonym* combination of a private synonym must be unique among all the synonyms in the database. That is, more than one private synonym with the same name can exist in the same database, but a different user must own each of these synonyms. The same user cannot create both a private and a public synonym that have the same name. For example, the following code generates an error:

```
CREATE SYNONYM our_custs FOR customer;
CREATE PRIVATE SYNONYM our_custs FOR cust_calls;-- ERROR!!!
```

A private synonym can be declared with the same name as a public synonym only if the two synonyms have different owners. If you own a private synonym, and a public synonym exists with the same name, the database server resolves the unqualified name as the private synonym. (In this case, you must specify *owner*.*synonym* to reference the public synonym.)

If you use DROP SYNONYM with the unqualified synonym identifier when your private synonym and the public synonym of another user both have the same identifier, only your private synonym is dropped. If you repeat the same DROP SYNONYM statement, the database server drops the public synonym.

Chaining Synonyms

About this task

If you create a synonym for a table or view that is not in the current database, and this table or view is dropped, the synonym remains registered in the system catalog. You can create a new synonym whose identifier is the name of the dropped table or view, but that points to a table or view in the current database (or in another database).

In this way, after you rename a table, or after you move a table or view to another database location, you can chain synonyms together so that the original synonym remains valid in existing applications. You can chain up to 16 synonyms in this manner.

Chaining synonyms to reference a relocated table object is possible for tables or views, but this is not valid for synonyms that point to a sequence object, because CREATE SYNONYM can define synonyms only for sequences that are registered in the current database.

The following steps chain two synonyms together for the **customer** table, which will ultimately reside on the **zoo** database server. Here ellipses (. . .) mark CREATE TABLE statements that are not complete:

1. In the **stores_demo** database on the database server that is called **training**, issue the following statement:

   ```
   CREATE TABLE customer (lname CHAR(15)...);
   ```

2. On the database server called **accntg**, issue the following statement:

   ```
   CREATE SYNONYM cust FOR stores_demo@training:customer;
   ```

3. On the database server called **zoo**, issue the following statement:

   ```
   CREATE TABLE customer (lname CHAR(15)...);
   ```

4. On the database server called **training**, issue the following statement:

   ```
   DROP TABLE customer;
   CREATE SYNONYM customer FOR stores_demo@zoo:customer;
   ```

The synonym **cust** on the **accntg** database server now points to the **customer** table on the **zoo** database server.

The following steps show an example of chaining two synonyms together and changing the table to which a synonym points:

1. On the database server called **training**, issue the following statement:

   ```
   CREATE TABLE customer (lname CHAR(15)...);
   ```

2. On the database server called **accntg**, issue the following statement:

   ```
   CREATE SYNONYM cust FOR stores_demo@training:customer;
   ```

3. On the database server called **training**, issue the following statement:

   ```
   DROP TABLE customer;
   CREATE TABLE customer (lastname CHAR(20)...);
   ```

The synonym **cust** on the **accntg** database server now points to a new version of the **customer** table on the **training** database server.

CREATE TABLE statement

Use the CREATE TABLE statement to create a new permanent table in the current database.

You can use the CREATE TABLE statement to create relational-database tables or to create typed tables (object-relational tables). For information about how to create temporary tables, see CREATE TEMP TABLE statement on page 370. For information about how to create external table objects that are not stored in the database, see CREATE EXTERNAL TABLE Statement on page 201.

Syntax

```
(explicit id unique_419_Connect_42_cret004) unique_419_Connect_42_cret004(explicit id unique_419_Connect_42_cret005)
unique_419_Connect_42_cret005(explicit id unique_419_Connect_42_cret007) unique_419_Connect_42_cret007(explicit id unique_419_Connect_42_cret009)
unique_419_Connect_42_cret009(explicit id unique_419_Connect_42_cret010) unique_419_Connect_42_cret010(explicit id unique_419_Connect_42_opt003)
unique_419_Connect_42_opt003(explicit id unique_419_Connect_42_opt004) unique_419_Connect_42_opt004(explicit id unique_419_Connect_42_opt005)
unique_419_Connect_42_opt005(explicit id unique_419_Connect_42_opt006) unique_419_Connect_42_opt006(explicit id unique_419_Connect_42_opt007)
unique_419_Connect_42_opt007(explicit id unique_419_Connect_42_opt008) unique_419_Connect_42_opt008(explicit id unique_419_Connect_42_opt009)
unique_419_Connect_42_opt009 CREATE { LARGE | SMALL } { STANDARD | RAW } TABLE [IF NOT EXISTS] table { ( <Column Definition>
[ { <Multiple-Column Constraint> | <Column Definition> } ] ) | [ <OF TYPE Clause> ] } [ [ WITH { AUDIT | CRCOLS | ERKEY
| REPLCHECK | VERCOLS } ] [ <SECURITY Clause> ] [ [ <Storage Options> ] [ LOCK MODE { PAGE | ROW } ] [ <USING Access-
Method > ] [ [ <Statistics Options> ] ]
```

Element	Description	Restrictions	Syntax
table	Name that you declare here for the new table	Must be unique among the names of tables, synonyms, views, and sequences in the database	Identifier on page 1231

Usage

When you create a table, you must declare its name and define its schema and its logging status. You can optionally specify additional attributes, as identified in topics that follow. The syntax diagram shows the sequence of required or optional specifications. These syntax segments of the CREATE TABLE statement, and some of their components, are identified in the five lists that follow.

The following keywords and clauses define column attributes of a new table:

Table 3. Defining the name, data-type, default value, and security label for a column

Specification	Topic	What the keyword or clause defines
Column Definition	Column definition on page 320	Column name and attributes, including data type, constraints, default value

Table 3. Defining the name, data-type, default value, and security label for a column (continued)

Specification	Topic	What the keyword or clause defines
DEFAULT	DEFAULT clause of CREATE TABLE on page 324	Default value for a column
COLUMN SECURED WITH	Column SECURED WITH label clause on page 322	An LBAC label for a protected column

The following keywords and clauses define constraints on the new table:

Table 4. Defining constraints on one or more columns of the table

Specification	Topic	What the keyword or clause defines
Single-Column Constraint	Single-Column Constraint Format on page 327	Data-integrity, referential, or other constraints on an individual column
Constraint Definition	Constraint Definition on page 336	Name, attributes, and enabled or disables status of constraints on the table
NULL	Using the NULL Constraint on page 329	Column allows NULL values
NOT NULL	Using the NOT NULL Constraint on page 328	Column does not allow NULL values
UNIQUE *or* DISTINCT	Using UNIQUE or DISTINCT Constraints on page 329	Column does not allow duplicate values
CHECK	CHECK Clause on page 335	Check constraints with other columns
PRIMARY KEY	Using the PRIMARY KEY Constraint on page 330	Contains a non-NULL unique value for each row in a table
FOREIGN KEY	Using the FOREIGN KEY Constraint on page 341	Establishes dependencies between tables
REFERENCES	REFERENCES Clause on page 331	Referential-integrity constraints with other columns
Multiple-Column Constraint	Multiple-Column Constraint Format on page 339	Data-integrity constraints on a set of columns

The following keywords and clauses define shadow columns and row-level audit support for the table:

Table 5. Defining shadow columns and row-level audit support

Specification	Topic	What the keyword or clause defines
WITH *keyword*	Specialized columns on page 346	Keyword options for shadow columns or for row-level audit support

Table 5. Defining shadow columns and row-level audit support (continued)

Specification	Topic	What the keyword or clause defines
WITH AUDIT	Using the WITH AUDIT Clause on page 346	Row-level audit support
WITH CRCOLS	Using the WITH CRCOLS Option on page 346	Keyword option for shadow columns or for row-level audit support
WITH ERKEY	Using the WITH ERKEY Keywords on page 347	3 shadow columns on which Enterprise Replication defines a primary key
WITH REPLCHECK	Using the WITH REPLCHECK Keywords on page 347	Shadow column that Enterprise Replication uses in consistency checking
WITH ROWIDS	Using the WITH ROWIDS Option on page 356	Hidden column in a fragmented table (deprecated)
WITH VERCOLS	Using the WITH VERCOLS Option on page 348	2 shadow columns for UPDATE operations on secondary servers

The following keywords and clauses define storage options for a new table:

Table 6. Defining storage for the table or for its smart-large-object columns

Specification	Topic	What the keyword or clause defines
Storage Options	Storage options on page 350	Where the table is physically stored and other information about how the table is stored
IN dbspace, sbspace, blobspace, *or* extspace	Using the IN Clause on page 351	Storage object to hold the new table (or part of it, or a large object)
FRAGMENT BY *or* PARTITION BY	FRAGMENT BY clause on page 356	Storage distribution scheme of a fragmented table
BY ROUND ROBIN	Fragmenting by ROUND ROBIN on page 357	A list of dbspaces for storing table fragments
BY EXPRESSION	Expression Fragment Clause on page 358	Expression-based fragment distribution
BY LIST	List fragment clause on page 361	List-based fragment distribution
BY RANGE . . . INTERVAL	#unique_87	RANGE INTERVAL-based fragment distribution
PUT Clause	PUT Clause on page 352	Storage location, extent size, and other sbspace attributes for a BLOB or CLOB column

Table 6. Defining storage for the table or for its smart-large-object columns (continued)

Specification	Topic	What the keyword or clause defines
EXTENT SIZE	EXTENT SIZE Options on page 363	Sizes of the first and subsequent storage extents of the table
COMPRESSED	COMPRESSED option for tables on page 365	Whether automatic compression of large amounts of row data is enabled

The following keywords and clauses define the logging mode and additional table attributes, or insert into the new table the qualifying rows that a specified query returns.

Table 7. Logging options, locking granularity, access methods, typed table attributes, data distribution statistics options, data insertion from query results, or an LBAC security policy for the table.

Specification	Topic	What the keyword or clause defines
Size Options (SMALL or LARGE)	SMALL and LARGE tables on page 319	Maximum number of pages in a new table
Logging Options (STANDARD or RAW)	Logging Options on page 319	Logging characteristics of the new table
LOCK MODE (PAGE *or* ROW)	LOCK MODE Options on page 365	Locking granularity of the new table
USING Access-Method	USING Access-Method Clause on page 367	How to access the new table
OF TYPE	OF TYPE Clause on page 343	Named ROW type of a typed table in an object-relational database
UNDER	Using the UNDER Clause on page 344	Supertable of a new subtable within a typed table hierarchy
SECURITY POLICY	SECURITY POLICY Clause on page 349	Label-based access control (LBAC) policy for the table
STATCHANGE, STATLEVEL	Statistics options of the CREATE TABLE statement on page 368	Change threshold and granularity of data distribution statistics
AS SELECT	AS SELECT clause on page 315	Creates and populates a query result table

Uniqueness rules for table names and column names

When you create a new table, every column must have a data type associated with it. The names of columns must be unique among the column in the same table. (The OF TYPE option specifies an existing named ROW type, whose fields provide column names and column data types for the typed table that you are creating.)

If the database was not created as MODE ANSI, the *table* name must be unique among all the identifiers of tables, views, sequence objects, and synonyms within the same database.

In an ANSI-compliant database, the combination *owner.table* must be unique among all the tables, synonyms, views, and sequence objects in the same database. Table objects qualified with different *owner* names can have the same identifier.

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if a table of the specified name is already registered in the current database.

Additional syntax notes for CREATE TABLE

For the restricted syntax options of CREATE TABLE statements that store the result set of a query in a new permanent table, see the AS SELECT clause on page 315.

In , using CREATE TABLE outside the CREATE SCHEMA statement generates warnings if you use the **-ansi** flag or if the **DBANSIWARN** environment variable is set.

The order of table options

The syntax diagram shows the order of table options in CREATE TABLE statements that include more than one of the following options:

- WITH options for specialized columns
- SECURITY POLICY options
- Storage options
- LOCK MODE options
- USING Access-Method clause
- Statistics options

For example, the following two CREATE TABLE statements are equivalent:

```
CREATE STANDARD TABLE IF NOT EXISTS myShadowy_tab(colA INT, colB CHAR)
   WITH ERRKEY, WITH CRCOLS, WITH AUDIT LOCK MODE ROW;

CREATE STANDARD TABLE IF NOT EXISTS myShadowy_tab(colA INT, colB CHAR)
   WITH AUDIT, WITH ERRKEY, WITH CRCOLS LOCK MODE ROW;
```

If you issue both statements consecutively in the same database, the second statement fails, because the table called **myShadowy_tab** that the first statement created already exists in the database. Because of the `IF NOT EXISTS` keywords, the redundant second statement returns no error, but it creates no new table.

The following example fails with an error, because no other Options clause can precede a WITH clause:

```
CREATE TABLE shadow_columns (colA INT, colB CHAR)
   LOCK MODE ROW WITH AUDIT, WITH ERRKEY, WITH CRCOLS; --incorrect options order
```

The next CREATE TABLE example also fails, because the Statistics option cannot precede the LOCK MODE option within the same Options clause:

```
CREATE TABLE shadow_columns (colA INT, colB CHAR)
   STATCHANGE 25 STATLEVEL TABLE LOCK MODE PAGE; --bad options order
```

AS SELECT clause

Use the AS SELECT clause of the CREATE TABLE statement to create a new table and to insert into it the data rows that are the result set of a specified query.

This syntax closely resembles in its functionality the INTO STANDARD and INTO RAW Clauses of the SELECT statement.

Only the following subset of the CREATE TABLE statement syntax is valid when you include the AS SELECT clause to create a new query result table, and to populate that table with the qualifying rows that the specified query returns:

(explicit id unique_469_Connect_42_qrt007) unique_469_Connect_42_qrt007(explicit id unique_469_Connect_42_qrt003)
unique_469_Connect_42_qrt003(explicit id unique_469_Connect_42_qrt004) unique_469_Connect_42_qrt004(explicit id unique_469_Connect_42_qrt005)
unique_469_Connect_42_qrt005

```
CREATE { STANDARD | [RAW] } TABLE [IF NOT EXISTS] [ owner . ] table [ ( { column | column_alias |
column::target_data_type } ) ] [ <Storage Options> ] [ <LOCK MODE> ] AS <SELECT clause>
```

Element	Description	Restrictions	Syntax
column	Column in a table in the FROM clause of the query. This will be a column name in the result table.	Must exist in the query result set	Identifier on page 1231
column_alias	Alias or display label for a column. This declares a column name in the result table.	Any nontrivial expression in the Projection clause of the query requires an alias or display label for its column name in the result table.	Identifier on page 1231
owner	Authorization identifier of the owner of the result table	Without this, the user issuing CREATE TABLE is *owner* by default	Owner name on page 1262
table	Name that you declare here for the result table	Must be unique among the names of tables, synonyms, views, and sequences in the database	Identifier on page 1231
target_data_type	Data type that is returned by an explicit cast. This will be the data type of a column in the result table.	See Rules for the Target Data Type on page 1024	Data Type on page 985

Usage

When using the CREATE TABLE . . . AS SELECT statement to create a new permanent table to store the result of a query, you can specify the logging mode of the table as STANDARD or RAW. If you omit both of these keywords, the default is STANDARD.

You can also optionally specify the following attributes for the query result table:

- A single storage location, or a distributed storage scheme
- The storage sizes of its first extent and next extent
- Its PAGE or ROW locking granularity

If you omit a storage or locking specification, the database server uses the default value.

If an error occurs while the database server is populating the new table with qualifying rows from the query in the AS SELECT clause, the operation is rolled back, and no new table is created or populated.

The columns that appear in the Projection list of the AS SELECT clause can be from any local table, or view, or from a remote database (which must be referenced in the qualified table name), but the new table must be created in the local database.

In a grid environment, if the tables in the FROM clause of the AS SELECT clause have the same schema in all the participating database servers of the specified grid or region, you can include the AS SELECT clause to create a result table from a grid query.

Supported data types

The CREATE TABLE . . . AS SELECT statement supports user-defined data types and all the built-in Informix® data types.

No more than one serial column, however, is allowed in the new result table. After the first column of type SERIAL, SERIAL8, or BIGSERIAL is included, any subsequent SERIAL, SERIAL8, or BIGSERIAL column is created as an INTEGER, INTEGER8, or BIGINTEGER column.

Column names in the result table

By default, the column names in the new permanent table are the names that are specified in the SELECT list of the Projection clause. If an asterisk (*) is the SELECT list of the Projection clause, the asterisk is expanded to all the column names in the corresponding tables or views in the FROM clause of the SELECT statement. Any explicit or implicit shadow columns in table objects specified by the FROM clause are not expanded by the asterisk specification.

On systems that implement Enterprise Replication, you can use the ADD CRCOLS, ADD REPLCHECK, and ADD ERKEY options to the ALTER TABLE statement to add the corresponding shadow columns to a result table that the AS SELECT clause creates.

All expressions in the SELECT list of the Projection clause, other than simple column expressions, must have a display label (also called a *column alias*). This is used as the identifier of the corresponding column in the new query result table. If a column expression has no display label, the result table uses the column name from the source table in the FROM clause of the query.

In the CREATE TABLE . . . AS SELECT statement, a column alias can be specified in either of two ways:

- As a comma-separated list of aliases, immediately following the TABLE keyword, similar to the syntax of the INSERT INTO . . . SELECT FROM statement
- As a part of the SELECT list in the Projection clause, just as in result tables that the SELECT . . . INTO STANDARD or SELECT . . . INTO RAW statements can create.

If both the SELECT list of the Projection clause and the comma-separated list of aliases that follows the TABLE keyword are present in the CREATE TABLE . . . AS SELECT statement, the comma-separated list of column aliases takes precedence. In this case, any column alias that you declare in the AS SELECT clause is ignored.

The CREATE TABLE . . . AS SELECT statement fails with an error in the following cases:

- If no display label or column alias is declared for a nontrivial column expression.
- If a display label or column alias has the same name as another column in the new result table.
- Except for storage options and LOCK MODE properties, the CREATE TABLE . . . AS SELECT statement cannot define constraints or any other special properties for columns of the new table.
- If a comma-separated list of aliases follows the TABLE keyword, but that list has fewer aliases than the number of expressions in the SELECT list of the Projection clause.

But if the column alias list that follows the TABLE keyword has more items than the SELECT list of the Projection clause, in this case the database server ignores the excess column aliases, and no exception occurs.

The AS SELECT clause can include a column in its ORDER BY clause that is not in the SELECT list of its Projection clause.

Restrictions on result tables

Besides restrictions that the section above identifies for display labels or column aliases and for unsupported column properties in result tables, the SELECT INTO . . . TABLE syntax of the SELECT statement is not valid as a part of a subquery.

As with most DDL statements, attempts to create the new result table in another database using the fully qualified table name fail with a syntax error. It is similarly an error to create a result table with the same name as an existing table, unless the AS SELECT clause includes the `IF NOT EXISTS` keywords, as described below.

IF NOT EXISTS keywords

If the name that you declare for the query result table in the AS SELECT clause is unique among the names of permanent tables, synonyms, views, and sequences in the database, the database server creates a query result table and populates it with all the qualifying rows that the query returns, whether or not the AS SELECT clause includes the `IF NOT EXISTS` keywords. (If the query returns no rows, the result table is empty, but its schema is registered in the system catalog of the database as a new permanent table.)

If the AS SELECT clause includes the `IF NOT EXISTS` keywords, and the name that you declare for the query result table is not unique among the names of permanent table objects in the database, the query that the AS SELECT clause defines is not executed, no result table is created, and the database server returns the message

```
0 row(s) retrieved into table.
```

If the AS SELECT clause omits the `IF NOT EXISTS` keywords, and the name that you declare for the query result table is not unique among the names of permanent table objects in the database, no result table is created, and the database server returns an error.

Examples of creating and populating result tables

The following example creates a new raw table called **rtabl** to store the results of a join query:

```
CREATE RAW TABLE IF NOT EXISTS rtab1
   AS
   SELECT t1col1, t1col2, t2col1
      FROM tab1, tab2
      WHERE t1col1 < 100 and t2col1 > 5;
```

In the example above, the new query result table **rtab1** would contain the columns **t1col1**, **t1col2** and **t2col1**.

The next example fails with error -249, because it declares no display label for the `col1+5` column expression:

```
CREATE TABLE IF NOT EXISTS qtab1
   AS
   SELECT col1+5, col2
      FROM tab1;
```

By declaring the column alias **qcol1** for the column expression in the Projection clause that includes the + operator, the following revised query avoids the -249 error that the previous example returns:

```
CREATE TABLE IF NOT EXISTS qtab1 (qcol1, col2)
   AS
   SELECT col1+5, col2
      FROM tab1;
```

The corrected example above creates the standard **qtabl** table to store the AS SELECT clause query results.

The next example uses different but equivalent aliasing syntax to declare the same **qcol1** alias in the AS SELECT clause, rather than in the list of column aliases:

```
CREATE TABLE IF NOT EXISTS qtab1
   AS
   SELECT col1+5 qcol1, col2
      FROM tab1;
```

The CREATE TABLE statement above similarly avoids the -249 error, and creates a result table that is identical in schema and in data content to the **qtabl** table in the previous example. In both of these examples, the result table has two columns, **qcol1** and **col2**. If **col1** is of type INTEGER, then **qcol1** would be type DECIMAL, the return data type from the expression `col1+5`.

As the syntax diagram indicates, the Storage and Lock Mode options to the CREATE TABLE statement are valid with the AS SELECT clause. The following example uses the `FRAGMENT BY EXPRESSION` keywords to define distributed storage for the query result table, where the **fcol1** column alias is the fragment key, and ROW is the locking granularity:

```
CREATE TABLE IF NOT EXISTS permtab (fcol1, col2)
   FRAGMENT BY EXPRESSION
      fcol1 < 300 IN dbs1,
      fcol1 >=300 IN dbs2
   LOCK MODE ROW
   AS SELECT col1::FLOAT, col2
         FROM tab1;
```

Any rows with **fcol1** values below `300` are inserted into dbspace **dbs1**. Rows with larger **fcol1** values are stored in the **dbs2** dbspace.

SMALL and LARGE tables

Use the SMALL keyword in the CREATE TABLE statement to create small, legacy-format tables in version 15.0.0.0.

Informix server version 15.0.0.0 supports significantly larger tables than prior versions, which had a limit of 16.7 million pages per fragment. The maximum page limit on page in version 15.0.0.0 is so much larger that these legacy tables are now referred to as *small tables*. Tables created in version 15.0.0.0 that may grow up to a maximum of 140 trillion pages per fragment are called *large tables*.

Large tables are made possible by new row addressing that uses an 8-byte identifier. Small tables use the legacy 4-byte row identifier.

A small table cannot be altered to become a large table using an ALTER TABLE statement. Nor may a large table be altered to a small table. Three supported methods of moving data from a small table to a large table are:

- unload/load
- selecting from the small table and inserting into the large table
- loopback replication

Indexes on large tables will have a larger footprint than indexes on small tables, since large table indexes contain the new 8-byte rowids. The percentage difference in footprint size will depend on the size of the index key and the number of duplicate keys. In terms of page numbers, the maximum size of an index on a large table is also significantly larger than the maximum size of an index on a small table.

By default, new permanent user tables in version 15.0.0.0 are large tables. It is possible to create small, legacy-format tables in version 15.0.0.0, however, using a new SMALL keyword in the CREATE TABLE statement. You may also change the default behaviour for a session by setting the TABLE_SIZE session-level environment variable to 'SMALL'.

System catalogs and tables in system databases (sysmaster, sysadmin, sysuser, sysutils, sysha, syscdr) are always small tables.

Temporary tables are always in small format unless created using the 'large' keyword. The TABLE_SIZE environment variable does not affect temporary tables.

While an instance is in *compatibility mode*, large tables may not be created.

Unless they are recompiled with the latest client libraries, legacy clients will not be able to access large tables, because they are not compatible with 8-byte rowids.

If a table is created with either the "SMALL" or "LARGE" keyword, dbschema and dbexport will preserve this keyword in the generated SQL as long as the **-ss** option is used. The schema of a table generated by dbschema -ss or dbexport -ss will not contain the "SMALL" or "LARGE" keyword unless it was used explicitly during the table creation.

Logging Options

Use the Logging Type options to specify logging characteristics that can improve performance in various bulk operations on the table.

Other than the default option (STANDARD) that is used for OLTP databases, these logging options are used primarily to improve performance in data warehousing databases.

A permanent table can have either of the following logging characteristics.

Logging Type	Effect
STANDARD	Logging tables that allow rollback, recovery, and restoration from archives. This type is the default. Use this type of table for all the recovery and constraints functionality that OLTP databases require.
RAW	Nonlogging tables that do not support primary key constraints or unique constraints. but that support referential constraints, and can be indexed and updated. Use this type of table for quickly loading data.



Warning: Use raw tables for fast loading of data, but set the logging type to STANDARD and perform a level-0 backup before you use the table in a transaction or modify the data within the table. If you must use a raw table within a transaction, either set the isolation level to Repeatable Read or lock the table in exclusive mode to prevent concurrency problems.

The CREATE RAW TABLE statement is not supported on secondary servers within a high-availability cluster.

For more information on these logging types of tables, refer to your *HCL® Informix® Administrator's Guide*.

Column definition

Use the column definition segment of the CREATE TABLE statement to declare the name and data type (and optionally the default value and the constraints or the security label) of a single column of the new table.

This syntax fragment is part of the CREATE TABLE statement on page 310.

(explicit id unique_444_Connect_42_coldef001a) unique_444_Connect_42_coldef001a(explicit id unique_444_Connect_42_coldef002a) unique_444_Connect_42_coldef002a(explicit id unique_444_Connect_42_coldef003a) unique_444_Connect_42_coldef003a(explicit id unique_444_Connect_42_coldef004ac) unique_444_Connect_42_coldef004ac

Column Definition

" *column* "

" <Data Type> [] "

" { [<DEFAULT Clause> []] [<Single-Column Constraint Format> []] | [<Single-Column Constraint Format> []] [<DEFAULT Clause> []] } "

" [<Column SECURED WITH label clause> []] "

Element	Description	Restrictions	Syntax
column	Name that you declare here for a column in the table	Must be unique in this table	Identifier on page 1231

Usage

Because the maximum row size is 32,767 bytes, no more than approximately 97 columns can be of COLLECTION data types (SET, LIST, and MULTISET). No more than approximately 195 columns in the table can be of the data types BYTE, TEXT, ROW, LVARCHAR, NVARCHAR, VARCHAR, and varying-length UDTs. (Here 195 columns is an approximate lower limit that applies to platforms with a 2 KB base page size. For platforms with a base page size of 4 KB, such as Windows™ and AIX® systems, the upper limit is approximately 450 columns of these data types.)

The upper limit on the number of columns of these data types also depends on other data that describes the table that the database server stores in the same partition. For some tables, the maximum number of columns might be lower, if the aggregate length of all the SQL identifiers (including the database name, table names, and index names) that are compressed and stored on the disk reduces the free space that is available for the columns.

Character column size semantics

Any explicit or default storage size specifications for columns of built-in character types, such as CHAR, LVARCHAR, NCHAR, NVARCHAR, or VARCHAR, are interpreted in units of bytes, unless the SQL_LOGICAL_CHAR configuration parameter is set to enable logical character semantics in data type declarations.

Interpreting size declarations as logical character semantics reduces the risk of insufficient storage for column values in INSERT and UPDATE operations When the data length exceeds the maximum size of the column, then the result depends on the ANSI-compliance status of the database:

- If the database is not ANSI-compliant, HCL Informix® truncates the value. No warning is generated when this truncation occurs.
- If the database is ANSI-compliant, then the INSERT or UPDATE operation fails and this error is returned:

```
-1279: Value exceeds string column length.
```

See the *HCL® Informix® Administrator's Reference* description of the SQL_LOGICAL_CHAR configuration parameter for more information about the effect of its setting in locales that support a multibyte code set, such as **UTF-8**, where a single logical character can require more than one byte of storage.

Restrictions on IDSSECURITYLABEL columns

The following restrictions affect the use of the Column Definition clause to specify a column of the IDSSECURITYLABEL data type to support label-based access control (LBAC):

- If the table has no security policy, a user who holds the DBSECADM role must also include the SECURITY POLICY clause to specify a security policy.
- Only a user who holds the DBSECADM role can specify a column of type IDSSECURITYLABEL.
- A table can have at most one column of type IDSSECURITYLABEL.
- The IDSSECURITYLABEL column cannot have column protection.
- The IDSSECURITYLABEL column has an implicit NOT NULL constraint by default. If no *label* name for the default security label is specified in the DEFAULT clause, the default value for this column is the security label for write access that is held by the user.
- The IDSSECURITY LABEL column cannot have any explicit single-column constraints, and it cannot be part of multiple-column referential or check constraints.
- The IDSSECURITYLABEL column cannot be encrypted.
- If the table is secured with both row-level and column-level protection, the default or explicit security label of the IDSSECURITYLABEL column must have the same security policy as any labels that the Column SECURED WITH clause references.

As with any SQL identifier, syntactic ambiguities (and sometimes error messages or unexpected behavior) can occur if the *column* name is a keyword, or if it is the same as the *table* name, or the name of another table that you later join with the *table*). For information about the keywords of Informix®, see Keywords of SQL for HCL Informix on page 1295.

If you define a column of a table as a named ROW type, the table does not adopt any constraints of the named ROW.

Column SECURED WITH label clause

Use the Column SECURED WITH label clause to provide label-based column-level security protection for a table by attaching a security label to the column. The label must be part of a security policy that the SECURITY POLICY clause attaches to the table.

This syntax fragment is part of the Column definition on page 320.

Column SECURED WITH label clause

" `[COLUMN] SECURED WITH` *label* "

Element	Description	Restrictions	Syntax
label	Name of a security label	Must exist and must belong to the security policy that protects the table.	Identifier on page 1231

Usage

The Column security clause can add label-based column-level protection. This clause is valid only for tables that are protected by a security policy. For the CREATE TABLE syntax to associate a label-based security policy with a table, see SECURITY POLICY Clause on page 349.

The user who includes the Column SECURED WITH label clause in the CREATE TABLE statement must hold the DBSECADAM role.

The security label can be the same label that protects other rows or columns of the table, or it can be a different label of the same security policy. The following restrictions apply to the `SECURED WITH` clause:

- The column cannot be of data type IDSSECURITYLABEL.
- You must specify the label without the policy qualifier, rather than as *policy.label*.
- The label must be a label of the security policy that secures the table.

Example of creating a table with column-level protection

The following CREATE TABLE statement defines a protected table called **Rigel** with the following schema:

```
CREATE TABLE Rigel IF NOT EXISTS Rigel
   (Col1 NCHAR(134)COLUMN SECURED WITH LabelRW,
   Col2 DATE,
   Col3 CHAR(20),
      SECURITY POLICY company;
```

The column security clause for column **Col1** provides column-level protection for the data stored in that column of the **Rigel** table by associating the security label **LabelRW** with column **Col1**. The SECURITY POLICY clause specifies the **company** security policy.

The CREATE TABLE statement in this example would fail if no security policy called **company** is defined in the database, or if the **company** policy exists, but no security label **LabelRW** is a component of the **company** security policy.

Example of creating a table with column-level and row-level protection

The following CREATE TABLE statement defines a protected table called **Vega** with the following schema:

```
CREATE TABLE Vega IF NOT EXISTS Vega
   (Col1 NCHAR(134)COLUMN SECURED WITH LabelRW,
   Col2 DATE,
   Col3 CHAR(20),
   Col4 IDSSECURITYLABEL DEFAULT LabelRW)
   SECURITY POLICY company;
```

The column security clause for column **Col1** provides column-level protection for the data stored in that column of the **Vega** table by associating the security label **LabelRW** with column **Col1**.

The table also has row-level protection from the **company** security policy, whose label **LabelRW** is the default value of the IDSSECURITYLABEL data type of **Col4**. In this example, label **LabelRW** provides row-level protection in **Col4**, and column-level protection in **Col1**.

The label stored in column **Col4** and the label securing column **Col1** could be different security labels, but both must be labels of the same **company** security policy.

The CREATE TABLE statement in this example would fail if no security policy called **company** is defined in the database, or if the **company** policy exists, but no security label **LabelRW** is a component of the **company** security policy.

DEFAULT clause of CREATE TABLE

Use the DEFAULT clause in the CREATE TABLE statement to specify the default value for the database server to insert into a column when no explicit value for the column is specified.

This syntax fragment is part of the Column definition on page 320.

(explicit id unique_163_Connect_42_defcl002) unique_163_Connect_42_defcl002(explicit id unique_163_Connect_42_defcl003)

unique_163_Connect_42_defcl003

DEFAULT Clause

" **DEFAULT** "

" { { **NULL** | *label* | *literal* | **USER** | **CURRENT_USER** } | [] { { **CURRENT** | **SYSDATE** } [<DATETIME Field Qualifier> []] | **TODAY** | **SITENAME** | **DBSERVERNAME** } } "

Element	Description	Restrictions	Syntax
label	Name of a security label	Must exist and must belong to the security policy that protects the table. The column must be of type IDSSECURITYLABEL.	Identifier on page 1231
literal	String of alphabetic or numeric characters	Must be an appropriate data type for the column. See Using a Literal as a Default Value on page 325.	Expression on page 1003

You cannot specify default values for SERIAL, BIGSERIAL, or SERIAL8 columns.

Using NULL as a default value

If you specify no default value for a column, the default is `NULL` unless you place a NOT NULL constraint on the column. In this case, no default value exists.

If you specify `NULL` as the default value for a column, you cannot specify a NOT NULL constraint as part of the column definition. (For details of NOT NULL constraints, see Using the NOT NULL Constraint on page 328.)

NULL is not a valid default value for a column that is part of a primary key.

For columns of large-object data types like BYTE, TEXT, BLOB, or CLOB, or of field-value pair data types like BSON or JSON, the only valid default value is `NULL`.

Example of setting the default value of a BSON column

The following CREATE TABLE statement fails with an exception, because the BSON data type of the **data** column requires `NULL` as its default, rather than the specified field-value pair:

```
CREATE TABLE tab1
  (
    id VARCHAR(128) NOT NULL,
    data "informix".BSON DEFAULT '"{id:1}"::JSON',
    modcount BIGINT,
    flags INTEGER DEFAULT 12,
    PRIMARY KEY (data)
  );

(U0001) - bson_to_char: unhandled storage type '98'
Error in line 8
```

The following successful statement replaces the non-NULL default value for **data** with the only valid default for BSON columns, and drops the PRIMARY KEY constraint on the **data** column.

```
CREATE TABLE tab1
  (
    id VARCHAR(128) NOT NULL,
    data "informix".BSON DEFAULT NULL,
    modcount BIGINT,
    flags INTEGER DEFAULT 12
  );
```

If this successful revision of the first example had defined a the PRIMARY KEY constraint on the **data** column, however, as in the first example, the CREATE TABLE statement would have failed with a different error, because for any data type, a column that has `NULL` as its default value cannot be part of a primary key.

Using a Literal as a Default Value

You can designate a literal value as a default value. A literal value is a string of alphabetic or numeric characters. To use a literal value as a default value, you must adhere to the syntax restrictions in the following table.

For Columns of Data Type	Format of Default Value
BOOLEAN	Use `'t'` or `'f'` (respectively for *true* or *false*) as a Quoted String on page 1202.
CHAR, CHARACTER VARYING, DATE, VARCHAR, NCHAR, NVARCHAR, LVARCHAR	Quoted String on page 1202. DATE literals must be of the format that the **DBDATE** (or else **GL_DATE**) environment variable specifies. In the default locale, if neither **DBDATE** nor **GL_DATE** is set, date literals must be of the *mm/dd/yyyy* format.
DATETIME	#unique_474
BIGINT, DECIMAL, FLOAT, INT8, INTEGER, MONEY, SMALLFLOAT, SMALLINT	Literal Number on page 1198
INTERVAL	#unique_475
Opaque data types	Quoted String on page 1202 in format of Single-Column Constraint Format on page 327

For example, the following statement includes a column definition with a literal DATETIME:

```
CREATE TABLE tab1
  (
    id VARCHAR(128) NOT NULL,
    date DATETIME YEAR TO FRACTION(3) DEFAULT DATETIME(1971-01-01 00:00:00.000)
         YEAR TO FRACTION(3),
    modcount BIGINT,
    flags INTEGER DEFAULT 12
  );
```

Using a Constant Expression as a Default Value

You can specify a constant expression as the default column value.

The following table lists constant expressions that you can specify, the data type requirements, and the recommended size (in bytes) for their corresponding columns.

Table 8. Constant expressions as default values

Constant Expression	Data Type Requirement	Recommended Size
CURRENT, SYSDATE	DATETIME column with matching qualifier	Enough bytes to store the longest DATETIME value for the locale
DBSERVERNAME, SITENAME	CHAR, VARCHAR, NCHAR, NVARCHAR, or CHARACTER VARYING column	128 bytes
TODAY	DATE column	Enough bytes to store the longest DATE value for the locale

Table 8. Constant expressions as default values (continued)

Constant Expression	Data Type Requirement	Recommended Size
USER, CURRENT_USER	CHAR, VARCHAR, NCHAR, NVARCHAR, or CHARACTER VARYING column	32 bytes

These column sizes are recommended because, if the column length is too small to store the default value during INSERT or ALTER TABLE operations, the database server returns an error.

You cannot designate a constant expression that behaves like a variant function (that is, CURRENT, SYSDATE, USER, TODAY, SITENAME, or DBSERVERNAME) as the default value for a column that holds an OPAQUE or DISTINCT data type. In addition, larger column sizes are required if the data values are encrypted, or if they are encoded in the Unicode character set of the **UTF-8** locale. (See the description of the SET ENCRYPTION statement later in this chapter for more information about storage size requirements for encrypted data.)

For descriptions of these functions, see Constant Expressions on page 1040.

The following example creates a table with columns that have literal default values. The **acc_id** column value defaults to the authorization identifier of the current user.

```
CREATE TABLE accounts (
   acc_num INTEGER DEFAULT 1,
   acc_type CHAR(1) DEFAULT 'A',
   acc_descr CHAR(20) DEFAULT 'New Account',
   acc_date DATETIME YEAR TO DAY DEFAULT SYSDATE DATETIME YEAR TO DAY,
   acc_id CHAR(32) DEFAULT CURRENT_USER);
```

Single-Column Constraint Format

Use the Single-Column Constraint format to define and declare the name of at least one constraint on a single column, and to specify the mode of each constraint.

The Single-Column Constraint format can associate one or more constraints with a column, in order to perform any of the following tasks:

- Create one or more data-integrity constraints for a column.
- Specify a meaningful name for a constraint.
- Specify the constraint-mode that controls the behavior of a constraint during insert, delete, and update operations.

This syntax fragment is part of the Column definition on page 320.

(explicit id unique_455_Connect_42_sccf002) unique_455_Connect_42_sccf002(explicit id unique_455_Connect_42_sccf003)

unique_455_Connect_42_sccf003(explicit id unique_455_Connect_42_sccf004) unique_455_Connect_42_sccf004(explicit id unique_455_Connect_42_sccf005)

unique_455_Connect_42_sccf005

Single-Column Constraint Format

```
" { { NULL | NOT NULL } | { [ DISTINCT | UNIQUE | PRIMARY KEY } | { <REFERENCES Clause> [ | <CHECK Clause> [ } } [
<Constraint Definition> [ ] "
```

The NULL constraint specifies that the column can store NULL values. It is not valid for columns of serial or complex data types. The CREATE TABLE statement fails with an error if you specify both NOT NULL and NULL constraints on the same column.

The following example creates a standard table with two constraints: **num**, a primary-key constraint on the **acc_num** column; and **code,** a unique constraint on the **acc_code** column:

```
CREATE TABLE accounts (
   acc_num   INTEGER PRIMARY KEY CONSTRAINT num,
   acc_code  INTEGER UNIQUE CONSTRAINT code,
   acc_descr CHAR(30));
```

The types of constraints used in this example are defined in sections that follow.

Restrictions on Using the Single-Column Constraint Format

The single-column constraint format cannot specify a constraint that involves more than one column. Thus, you cannot use the single-column constraint format to define a composite key. For information on multiple-column constraints, see Multiple-Column Constraint Format on page 339.

You cannot define a referential constraint or a unique constraint on any column of a RAW table. Only NOT NULL or NULL constraints are supported on RAW tables.

You cannot place unique, primary-key, or referential constraints on BLOB, BYTE, CLOB, or TEXT columns. You can, however, check for NULL or non-NULL values on BYTE or TEXT columns with a check constraint.

If the constraint is on a column that stores encrypted data, Informix® cannot enforce the constraint.

Using the NOT NULL Constraint

Use the NOT NULL keywords to require that a column receive a value during insert or update operations. If you place a NOT NULL constraint on a column (and no default value is specified), you *must* enter a value into this column when you insert a row or update that column in a row. If you do not enter a value, the database server returns an error, because no default value exists.

The following example creates the **newitems** table. In **newitems**, the column **manucode** does not have a default value nor does it allow NULL values.

```
CREATE TABLE newitems (
   newitem_num INTEGER,
   manucode CHAR(3) NOT NULL,
   promotype INTEGER,
   descrip CHAR(20));
```

When you define a PRIMARY KEY constraint, the database server also silently creates a NOT NULL constraint on the same column, or on the same set of columns that make up the primary key.

You cannot specify NULL as the explicit default value for a column if you also specify the NOT NULL constraint.

The CREATE TABLE statement fails with an error if you specify both a NOT NULL constraint and a NULL constraint on the same column.

The NOT NULL constraint is required for columns of the collection data types LIST, MULTISET, and SET. No other column constraints are allowed on a collection data type.

Using the NULL Constraint

Use the NULL keyword to specify that a column can store the NULL value for its data type. This implies that the column need not receive any value during insert or update operations. The NULL constraint is logically equivalent to omitting the NOT NULL constraint from the column definition.

The following example creates the **newitems** table. In **newitems**, the column **descrip** does not have a default value, but it allows NULL values.

```
CREATE TABLE newitems (
   newitem_num INTEGER,
   manucode CHAR(3) NOT NULL,
   promotype INTEGER,
   descrip CHAR(20) NULL);
```

In the example above, the columns **newitem_num** and **promotype** also allow NULL values implicitly, because no NOT NULL constraint is defined on them.

The CREATE TABLE statement fails with an error if you specify both a NOT NULL constraint and a NULL constraint on the same column.

You cannot specify both a NULL constraint and a PRIMARY KEY constraint on the same column, because when the CREATE TABLE statement defines a PRIMARY KEY constraint, the database server also silently creates a NOT NULL constraint on the same column, or on the same set of columns that make up the primary key.

The NULL constraint is not valid for columns of the collection data types LIST, MULTISET, and SET, nor for IDSSECURITYLABEL columns.

Using UNIQUE or DISTINCT Constraints

Use the UNIQUE or DISTINCT keyword to require that a column or set of columns accepts only unique data values. You cannot insert values that duplicate the values of some other row into a column that has a unique constraint. When you create

a UNIQUE or DISTINCT constraint, the database server automatically creates an internal index on the constrained column or columns. (In this context, the keyword DISTINCT is a synonym for UNIQUE.)

You cannot place a unique constraint on a column that already has a primary-key constraint. You cannot place a unique constraint on a BYTE or TEXT column.

As previously noted, you cannot place a unique or primary-key constraint on a BLOB or CLOB column of Informix®.

Opaque data types support a unique constraint only where a secondary-access method supports uniqueness for that type. The default secondary-access method is a generic B-tree, which supports the **equal( )** operator function. Therefore, if the definition of the opaque type includes the **equal( )** function, a column of that opaque type can have a unique constraint.

The following example creates a simple table that has a unique constraint on one of its columns:

```
CREATE TABLE accounts
   (acc_name  CHAR(12),
    acc_num   SERIAL UNIQUE CONSTRAINT acc_num);
```

For an explanation of the constraint name, refer to Declaring a Constraint Name on page 336.

Differences Between a Unique Constraint and a Unique Index

Although a unique index and a unique constraint are functionally similar, besides various differences in the syntax by which you declare, alter, or destroy them, there are additional differences between these two types of database objects:

- In DDL statements, they are registered or dropped in different tables of the system catalog
- In DML statements, enabled unique constraints on a logged table are checked at the end of a statement, but unique indexes are checked on a row-by-row basis, thereby preventing any insert or update of a row that might potentially violate the uniqueness of the specified column (or for a multiple-column column constraint or index, the column list).

For example, if you stored the values 1, 2, and 3 in rows of a logged table that has an INT column, an UPDATE operation on that table that specifies `SET c = c + 1` would fail with an error if there were a unique index on the column `c`, but the statement would succeed if the column had a unique constraint.

Using the PRIMARY KEY Constraint

A *primary key* is a column (or a set of columns, if you use the multiple-column constraint format) that contains a non-NULL, unique value for each row in a table. When you define a PRIMARY KEY constraint, the database server automatically creates an internal index on the column or columns that make up the primary key, and silently creates a NOT NULL constraint on the same column or columns.

You can designate only one primary key for a table. If you define a single column as the primary key, then it is unique by definition. You cannot explicitly give the same column a unique constraint.

You cannot place a unique or primary-key constraint on a BLOB or CLOB column.

Opaque types of Informix® support a primary key constraint only where a secondary-access method supports the uniqueness for that type. The default secondary-access method is a generic B-tree, which supports the **equal( )** function.

Therefore, if the definition of the opaque type includes the **equal( )** function, a column of that opaque type can have a primary-key constraint.

You cannot place a primary-key constraint on a BYTE or TEXT column.

In the previous two examples, a unique constraint was placed on the column **acc_num**. The following example creates this column as the primary key for the **accounts** table:

```
CREATE TABLE accounts
   (acc_name  CHAR(12),
    acc_num   SERIAL PRIMARY KEY CONSTRAINT acc_num);
```

REFERENCES Clause

Use the REFERENCES clause to establish a referential relationship:

- Within a table (that is, between two columns of the same table)
- Between two tables (in other words, create a foreign key)

This syntax fragment is part of the Single-Column Constraint Format on page 327 and the Multiple-Column Constraint Format on page 339.

(explicit id unique_462_Connect_42_refcl002) unique_462_Connect_42_refcl002

REFERENCES Clause

" **REFERENCES** *table* "

" [(*column*)] "

" [[] **ON DELETE CASCADE**] "

Element	Description	Restrictions	Syntax
column	A referenced column	See Restrictions on Referential Constraints on page 332.	Identifier on page 1231
table	The referenced table	Must reside in the same database as the referencing table	Identifier on page 1231

The *referencing* column (the column being defined) is the column or set of columns that refers to the referenced column or set of columns. The referencing column can contain NULL and duplicate values, but values in the referenced column (or set of columns) must be unique.

The relationship between referenced and referencing columns is called a *parent-child* relationship, where the parent is the referenced column (primary key) and the child is the referencing column (foreign key). The referential constraint establishes this parent-child relationship.

When you create a referential constraint, the database server automatically creates an internal index on the constrained column or columns.

Restrictions on Referential Constraints

You must have the References privilege to create a referential constraint.

When you use the REFERENCES clause, you must observe the following restrictions:

- The referenced and referencing tables must be in the same database.
- The referenced column (or set of columns when you use the multiple-column constraint format) must have a unique or primary-key constraint.
- The data types of the referencing and referenced columns must be identical.

  The only exceptions are that a referencing column must be an integer data type if the referenced column is a serial data type:
  - For BIGSERIAL referenced columns, use BIGINT referencing columns.
  - For SERIAL referenced columns, use INT referencing columns.
  - For SERIAL8 referenced columns, use INT8 referencing columns.
- You cannot place a constraint on any column of a RAW table.
- You cannot place a referential constraint on a BYTE, TEXT, BLOB, or CLOB column.
- When you use the single-column constraint format, you can reference only one column.
- When you use the multiple-column constraint format, the maximum number of columns in the REFERENCES clause is 16, and the total length of the columns cannot exceed 390 bytes if the page size is 2 kilobytes. (The maximum length increases with the page size.)

Default Values for the Referenced Column

If the referenced table is different from the referencing table, you do not need to specify the referenced column; the default column is the primary-key column (or columns) of the referenced table. If the referenced table is the same as the referencing table, you must specify the referenced column.

Referential Relationships Within a Table

You can establish a referential relationship between two columns of the same table. In the following example, the **emp_num** column in the **employee** table uniquely identifies every employee through an employee number. The **mgr_num** column in that table contains the employee number of the manager who manages that employee. In this case, **mgr_num** references **emp_num**. Duplicate values appear in the **mgr_num** column because managers manage more than one employee.

```
CREATE TABLE employee
   (
   emp_num INTEGER PRIMARY KEY,
   mgr_num INTEGER REFERENCES employee (emp_num)
   );
```

A table in which referential relationships exist among its rows can have a PRIMARY KEY constraint with no explicit foreign key. For the syntax to recursively query a table in which multiple levels of a logical hierarchy exist among the rows, see Hierarchical Clause on page 738.

Locking Implications of Creating a Referential Constraint

When you create a referential constraint, an exclusive lock is placed on the referenced table. The lock is released when the CREATE TABLE statement is finished. If you are creating a table in a database that supports transaction logging, and you are using transactions, the lock is released at the end of the transaction.

Examples of the Single-Column Constraint format

These examples illustrate single-column constraint format options to define a foreign-key constraint that is enabled by default, and to declare the name of a disabled referential constraint.

A referential constraint enabled by default

The following example uses the single-column constraint format to define a referential relationship between the sub_accounts and accounts tables. (The terms *foreign-key constraint* and *referential constraint* are synonyms.) The **ref_num** column (the foreign key) in the sub_accounts table references the acc_num column (the primary key) in the accounts table.

```
CREATE TABLE accounts (
   acc_num INTEGER PRIMARY KEY,
   acc_type INTEGER,
   acc_descr CHAR(20));
CREATE TABLE sub_accounts (
   sub_acc INTEGER PRIMARY KEY,
   ref_num INTEGER REFERENCES accounts (acc_num),
   sub_descr CHAR(20));
```

The single-column constraint format syntax of the CREATE TABLE statement above that defines the sub_accounts table does not explicitly specify that the ref_num column is a foreign key, but the `REFERENCES` keyword specifies that ref_num must have the same value as the acc_num column in some row of the accounts table. This implies that the ref_num column is the foreign key in a referential relationship in which sub_accounts is the referencing table, and accounts is the referenced table.

In single-column constraint format, you do not explicitly specify that the ref_num column is a foreign key. To include the `FOREIGN KEY` keywords when you place a referential constraint on a single column (or on a list of columns that reference the same primary key) of the referencing table, you must instead use the multiple-column constraint format syntax to define the referential constraint.

By default, this constraint on the sub_accounts table is enabled without filtering, because no explicit constraint mode is specified. You can use the neither the `DISABLED` or `FILTERING` keyword is specified in the example. The SET CONSTRAINTS option to the SET Database Object Mode statement can reset the object mode of existing constraints.

Because the sub_accounts example above declares no name for the referential constraint, the database server generates an implicit identifier when it registers this constraint in the sysconstraints system catalog table, and registers its mode (`E`) in the sysobjstate system catalog table.

A disabled referential constraint

The next CREATE TABLE statement creates a xeno_counts table, and defines a referential constraint between its xeno_num column and the acc_num column in the accounts table from the first example. This single-column constraint format syntax also includes a constraint definition, specifying `DISABLED` as its constraint mode, and declaring xeno_constr as the name of this foreign-key constraint. Here xeno_accounts is the referencing table, and accounts is the referenced table.

```
CREATE TABLE xeno_counts (
   xeno_acc INTEGER PRIMARY KEY,
   xeno_num INTEGER REFERENCES accounts (acc_num)
      CONSTRAINT xeno_constr DISABLED,
   xeno_descr CHAR(20));
```

In `DISABLED` mode, the xeno_constr constraint is not enforced when DML operations produce violating rows in the xeno_counts table. To enforce referential integrity, however, you can use the SET CONSTRAINTS option to the SET Database Object Mode statement to change the constraint mode to `ENABLED`. Alternatively, SET CONSTRAINTS can reset the xeno_constr constraint to a `FILTERING` mode, after the START VIOLATIONS statement associates a violations table with the xeno_counts table.

Using the ON DELETE CASCADE Option

Use the ON DELETE CASCADE option to specify whether you want rows deleted in a child table when corresponding rows are deleted in the parent table. If you do not specify cascading deletes, the default behavior of the database server prevents you from deleting data in a table if other tables reference it.

If you specify this option, later when you delete a row in the parent table, the database server also deletes any rows associated with that row (foreign keys) in a child table. The principal advantage to the cascading-deletes feature is that it allows you to reduce the quantity of SQL statements you need to perform delete actions.

For example, the **all_candy** table contains the **candy_num** column as a primary key. The **hard_candy** table refers to the **candy_num** column as a foreign key. The following CREATE TABLE statement creates the **hard_candy** table with the cascading-delete option on the foreign key:

```
CREATE TABLE all_candy
   (candy_num SERIAL PRIMARY KEY,
    candy_maker CHAR(25));

CREATE TABLE hard_candy
   (candy_num INT,
    candy_flavor CHAR(20),
    FOREIGN KEY (candy_num) REFERENCES all_candy
    ON DELETE CASCADE);
```

Because ON DELETE CASCADE is specified for the dependent table, when a row of the **all_candy** table is deleted, the corresponding rows of the **hard_candy** table are also deleted. For information about syntax restrictions and locking implications when you delete rows from tables that have cascading deletes, see Considerations When Tables Have Cascading Deletes on page 454.

CHECK Clause

Use the CHECK clause to designate conditions that must be met *before* data can be assigned to a column during an INSERT or UPDATE statement.

This syntax fragment is part of the Single-Column Constraint Format on page 327 and the Multiple-Column Constraint Format on page 339.

(explicit id unique_459_Connect_42_check001) unique_459_Connect_42_check001

CHECK Clause

" **CHECK** "

" (<Condition> []) "

The *condition* cannot include a user-defined routine.

During an insert or update, if the check constraint of a row evaluates to *false*, the database server returns an error. The database server does not return an error if a row evaluates to NULL for a check constraint. In some cases, you might want to use both a check constraint and a NOT NULL constraint.

Using a Search Condition

The *search condition* that defines a check constraint cannot contain the following elements: user-defined routines, subqueries, aggregates, host variables, or rowids. In addition, the search condition cannot contain the following built-in variant functions: CURRENT, SYSDATE, USER, CURRENT_USER, SITENAME, DBSERVERNAME, or TODAY.

When you specify a date value in a search condition, make sure you specify four digits for the year, so that the **DBCENTURY** environment variable has no effect on the condition. When you specify a two-digit year, the **DBCENTURY** environment variable can produce unpredictable results if the condition depends on an abbreviated year value. For more information about **DBCENTURY**, see the *HCL® Informix® Guide to SQL: Reference*.

More generally, the database server saves the settings of environment variables from the time of creation of check constraints. If any of these settings are subsequently changed in a way that can affect the evaluation of a condition in a check constraint, the new settings are disregarded, and the original environment variable settings are used when the condition is evaluated.

With a BYTE or TEXT column, you can check for NULL or not-NULL values. This constraint is the only constraint allowed on a BYTE or TEXT column.

Restrictions When Using the Single-Column Constraint Format

When you use the single-column constraint format to define a check constraint, the check constraint cannot depend on values in other columns of the table. The following example creates the **my_accounts** table that has two columns with check constraints, each in the single-column constraint format:

```
CREATE TABLE my_accounts (
   chk_id   SERIAL PRIMARY KEY,
   acct1    MONEY CHECK (acct1 BETWEEN 0 AND 99999),
   acct2    MONEY CHECK (acct2 BETWEEN 0 AND 99999));
```

Both **acct1** and **acct2** are columns of MONEY data type whose values must be between `0` and `99999`. If, however, you want to test that **acct1** has a larger balance than **acct2**, you cannot use the single-column constraint format. To create a constraint that checks values in more than one column, you must use the Multiple-Column Constraint Format on page 339.

Constraint Definition

Use the constraint definition portion of CREATE TABLE for these purposes:

- To declare a name for the constraint
- To set a constraint to disabled, enabled, or filtering mode.

This syntax fragment is part of the Single-Column Constraint Format on page 327 and the Multiple-Column Constraint Format on page 339.

Constraint Definition

" [**CONSTRAINT** *constraint*] "

" [[{ **ENABLED** | **DISABLED** | **FILTERING** { **WITHOUT ERROR** | **WITH ERROR** } }]] "

Element	Description	Restrictions	Syntax
constraint	Name of constraint	Must be unique for the table among index and constraint names	Identifier on page 1231

Declaring a Constraint Name

The database server implements the constraint as an index. Whenever you use the single- or multiple-column constraint format to place a data restriction on a column, but without declaring a *constraint* name, the database server creates a constraint and adds a row for that constraint in the **sysconstraints** system catalog table.

The database server also generates an identifier and adds a row to the **sysindices** system catalog table for each new primary-key, unique, or referential constraint that does not share an index with an existing constraint. Even if you declare a

name for a constraint, the database server generates the name that appears in the **sysindices** table. (The system catalog also includes a view on the **sysindices** table, called **sysindexes**, which also lists each component of a composite index.)

If you want, you can specify a meaningful name for the constraint. The name must be unique among the names of constraints and indexes in the database.

Constraint names appear in error messages having to do with constraint violations. You can use this name when you use the DROP CONSTRAINT clause of the ALTER TABLE statement.

You also specify a constraint name when you change the mode of constraint with the SET Database Object Mode statement or the SET Transaction Mode statement, and in the DROP INDEX statement for constraints that are implemented as indexes with user-defined names.

In an ANSI-compliant database, when you declare the name of a constraint of any type, the combination of the *owner* name and *constraint* name must be unique within the database.

Constraint Names That the Database Server Generates

If you do not specify a constraint name, the database server generates a constraint name using the following template:

```
constraint_typetabid_constraintid
```

In this template, *constraint_type* is the letter **u** for unique or primary-key constraints, **r** for referential constraints, **c** for check constraints, and **n** for NOT NULL constraints. In the template, *tabid* and *constraintid* are values from the **tabid** and **constrid** columns of the **systables** and **sysconstraints** system catalog tables, respectively. For example, the constraint name for a unique constraint might look like ” **u111_14**” (with a leading blank space).

If the generated name conflicts with an existing identifier, the database server returns an error, and you must then supply an explicit constraint name.

The generated index name in **sysindexes** (or **sysindices**) has this format:

```
[blankspace]tabid__constraintid
```

For example, the index name might be something like " **111_14** " (quotation marks used here to show the blank space).

Choosing a Constraint-Mode Option

Use the constraint-mode options (ENABLED, DISABLED, and FILTERING) to control the behavior of constraints in INSERT, DELETE, MERGE, and UPDATE operations.

For constraints that the CREATE TABLE statement defines, these are the options.

DISABLED

Does not enforce the constraint during INSERT, DELETE, and UPDATE operations

ENABLED

Enforces the constraint during INSERT, DELETE, and UPDATE operations If a target row causes a violation of the constraint, the statement fails. This mode is the default.

FILTERING

Enforces the constraint during INSERT, DELETE, and UPDATE operations, if the START VIOLATIONS statement has created a violations table and a diagnostics table. If a target row causes a violation of the constraint, the statement continues processing. The database server writes the row in question to the violations table associated with the target table, and writes diagnostic information to the associated diagnostics table.

If you choose filtering mode, you can specify the WITHOUT ERROR or WITH ERROR options. The following list explains these ERROR options.

WITH ERROR

Returns an integrity-violation error when a filtering-mode constraint is violated during an INSERT, DELETE, or UPDATE operation.

WITHOUT ERROR

Does not return an integrity-violation error when a filtering-mode constraint is violated during an INSERT, DELETE, or UPDATE operation. This is the default ERROR option.



Note:

For the FILTERING WITHOUT ERROR mode to have these effects, you must also use the START VIOLATIONS TABLE statement to start the violations and diagnostics tables for the target table on which the constraints are defined. You can issue that statement

- either before you set any constraints on the table to a filtering mode,
- or after you set constraints to a filtering mode, but before any users perform INSERT, DELETE, or UPDATE operations on rows in the table.

Constraint modes are registered in the sysobjstate system catalog table.

NOVALIDATE modes for foreign-key or check constraints

The modes listed above are only a subset of the constraint modes that the SET CONSTRAINTS option to the SET Database Object Mode statement can specify while it is resetting the mode of an existing foreign-key or check constraint. They are also a subset of the constraint modes that the ALTER TABLE ADD CONSTRAINT statement can specify while creating a new foreign-key or check constraint on an existing table.

The ALTER TABLE ADD CONSTRAINT and SET CONSTRAINTS statements can specify one of these additional constraint modes by including the `NOVALIDATE` keyword in the constraint definition. The effect is that the database server skips the checking of existing rows for violations when the constraint is being created or enabled, thereby reducing the time and resources required for processing the DDL statement. When that statement completes execution, however, each NOVALIDATE mode automatically reverts to an ENABLED or FILTERING mode. Thus, the `NOVALIDATE` keyword does not prevent enforcement of referential integrity or check conditions during subsequent DML operations on the table, because the NOVALIDATE modes do not persist beyond the DDL statement that defined them.

Because most tables are empty when they are created, referential-integrity checking of existing rows typically does not occur during table creation, and the CREATE TABLE statement does not support NOVALIDATE constraint modes. Those modes can be efficient, however, in contexts where non-empty tables with foreign-key or check constraints need to be moved to another database or to a data warehouse.

Multiple-Column Constraint Format

Use the multiple-column constraint format to associate one or more columns with a constraint. This alternative to the single-column constraint format allows you to associate multiple columns with a constraint.

This syntax fragment is part of the CREATE TABLE statement on page 310 and the OF TYPE Clause on page 343.

(explicit id unique_445_Connect_42_mccf002) unique_445_Connect_42_mccf002(explicit id unique_445_Connect_42_mccf003)

unique_445_Connect_42_mccf003(explicit id unique_445_Connect_42_mccf004) unique_445_Connect_42_mccf004(explicit id

unique_445_Connect_42_mccf005) unique_445_Connect_42_mccf005

Multiple-Column Constraint Format

" { { **NOT NULL** | **NULL** } { **UNIQUE** | [**DISTINCT** | **PRIMARY KEY** | <REFERENCES Clause>] } (*column*) | **FOREIGN KEY** (*column*) <REFERENCES Clause> [| <CHECK Clause>] } "

" [<Constraint Definition>] "

Element	Description	Restrictions	Syntax
column	Columns on which to place constraint	Not BYTE, TEXT, BLOB, CLOB	Identifier on page 1231

A multiple-column constraint has these cardinality and size restrictions:

- It can specify no more than 16 column names.
- In Informix®, the maximum total length of the list of columns depends on the page size:
  - For page size of 2K, the total length cannot exceed 380 bytes.
  - For page size of 16K, the total length cannot exceed 3247 bytes.
  - A page size of 256K will support a total length of 32K, which is the maximum row size.

Here the slash (/) symbol represents integer division.

When you define a unique constraint (by using the UNIQUE or DISTINCT keyword), a column cannot appear in the constraint list more than once.

Using the multiple-column constraint format, you can perform these tasks:

- Create data-integrity constraints for a set of one or more columns
- Declare a mnemonic name for a constraint
- Specify the constraint-mode option that controls the behavior of a constraint during insert, delete, and update operations.

When you use this format, you can create composite primary and foreign keys, or define check constraints that compare data in different columns.

See also the section Differences Between a Unique Constraint and a Unique Index on page 330.

Restrictions with the Multiple-Column Constraint Format

When you use the multiple-column constraint format, you cannot define any default values for columns. In addition, you cannot establish a referential relationship between two columns of the same table.

To define a default value for a column, or to establish a referential relationship between two columns of the same table, refer to Single-Column Constraint Format on page 327 or to Referential Relationships Within a Table on page 332 respectively.

Using Large-Object Types in Constraints

You cannot place unique, primary-key, or referential (FOREIGN KEY) constraints on BYTE or TEXT columns. You can, however, check for NULL or non-NULL values with a check constraint.

You cannot place unique or primary-key constraints on BLOB or CLOB columns. If the constraint is on a set of columns that includes a column that stores encrypted data, Informix® cannot enforce the constraint.

You can find detailed discussions of specific constraints in the following sections:

Constraint	For more information, see	For an example, see
CHECK	CHECK Clause on page 335	Defining Check Constraints Across Columns on page 341
DISTINCT	Using UNIQUE or DISTINCT Constraints on page 329	Examples of the Multiple-Column Constraint Format on page 341
FOREIGN KEY	Using the FOREIGN KEY Constraint on page 341	Defining Composite Primary and Foreign Keys on page 341
PRIMARY KEY	Using the PRIMARY KEY Constraint on page 330	Defining Composite Primary and Foreign Keys on page 341
UNIQUE	Using UNIQUE or DISTINCT Constraints on page 329	Examples of the Multiple-Column Constraint Format on page 341

Using the FOREIGN KEY Constraint

A foreign key *joins* and establishes dependencies between tables. That is, it creates a referential constraint. (For more information on referential constraints, see the REFERENCES Clause on page 331.)

A foreign key references a unique or primary key in a table. For every entry in the foreign-key columns, a matching entry must exist in the unique or primary-key columns if all foreign-key columns contain non-NULL values.

You cannot specify BYTE or TEXT columns as foreign keys.

You cannot specify BLOB or CLOB columns as foreign keys.

Examples of the Multiple-Column Constraint Format

The following example creates a standard table, called **order_items**, with a unique constraint, called **items_constr**, using the multiple-column constraint format:

```
CREATE TABLE order_items
   (
   order_id SERIAL,
   line_item_id INT not null,
   unit_price DECIMAL(6,2),
   quantity INT,
   UNIQUE (order_id,line_item_id) CONSTRAINT items_constr
   );
```

For constraint names, see Declaring a Constraint Name on page 336.

Defining Check Constraints Across Columns

When you use the multiple-column constraint format to define check constraints, a check constraint can apply to more than one column in the same table. (You cannot, however, create a check constraint whose *condition* uses a value from a column in another table.)

This example compares two columns, **acct1** and **acct2**, in the new table:

```
CREATE TABLE my_accounts
   (
   chk_id   SERIAL PRIMARY KEY,
   acct1    MONEY,
   acct2    MONEY,
   CHECK (0 < acct1 AND acct1 < 99999),
   CHECK (0 < acct2 AND acct2 < 99999),
   CHECK (acct1 > acct2)
   );
```

In this example, the **acct1** column must be greater than the **acct2** column, or the insert or update fails.

Defining Composite Primary and Foreign Keys

When you use the multiple-column constraint format, you can create a composite key. A *composite key* specifies multiple columns for a primary-key or foreign-key constraint.

The next example creates two tables. The first table has a composite key that acts as a primary key, and the second table has a composite key that acts as a foreign key.

```
CREATE TABLE accounts (
   acc_num INTEGER,
   acc_type INTEGER,
   acc_descr CHAR(20),
   PRIMARY KEY (acc_num, acc_type));

CREATE TABLE sub_accounts (
   sub_acc INTEGER PRIMARY KEY,
   ref_num INTEGER NOT NULL,
   ref_type INTEGER NOT NULL,
   sub_descr CHAR(20),
   FOREIGN KEY (ref_num, ref_type) REFERENCES accounts
      (acc_num, acc_type));
```

In this example, the foreign key of the **sub_accounts** table, **ref_num** and **ref_type**, references the composite key, **acc_num** and **acc_type**, in the **accounts** table. If, during an insert or update, you tried to insert a row into the **sub_accounts** table whose value for **ref_num** and **ref_type** did not exactly correspond to the values for **acc_num** and **acc_type** in an existing row in the **accounts** table, the database server would return an error.

A referential constraint must have a one-to-one relationship between referencing and referenced columns. In other words, if the primary key is a set of columns (a composite key), then the foreign key also must be a set of columns that corresponds to the composite key.

Because of the default behavior of the database server, when you create the foreign-key reference, you do not need to reference the composite-key columns (**acc_num** and **acc_type**) explicitly. You can rewrite the references section of the previous example as follows:

```
FOREIGN KEY (ref_num, ref_type) REFERENCES accounts
```

Default Index Creation Strategy for Constraints

When you create a table with unique or primary-key constraints, the database server creates a system-generated, internal index that is unique and ascending for each constraint.

When you drop a primary-key or foreign-key constraint, the corresponding system-generated index is also dropped. However, you can preserve the index by renaming it with the RENAME INDEX statement.

When you create a table with a referential constraint, the database server creates an internal index that is ascending and that allows duplicate values for each column that you specify in the referential constraint.

An internal index occupies the same storage location as its table. For fragmented tables, the fragments of an internal index occupy the same dbspace partitions that you specify for the table fragments (or in some cases, the database dbspace).

If you require an index fragmentation strategy that is independent of the underlying table fragmentation, do not define the constraint when you create the table. Instead, use the CREATE INDEX statement to create a unique index with the desired fragmentation strategy. Then use the ALTER TABLE statement to add the constraint. The new constraint uses the previously defined index.

Important: In a database without logging, *detached checking* is the only kind of constraint checking available. Detached checking means that constraint checking is performed on a row-by-row basis.

OF TYPE Clause

Use the OF TYPE clause to create a *typed table* for an object-relational database. A typed table is a table in which every row is an object of the named ROW data type that you specify in this clause.

This syntax fragment is part of the CREATE TABLE statement on page 310.

(explicit id unique_446_Connect_42_otcl003) unique_446_Connect_42_otcl003(explicit id unique_446_Connect_42_otcl004) unique_446_Connect_42_otcl004

OF TYPE Clause

" **OF TYPE** *row_type* "

" [(<Multiple-Column Constraint Format> [])] "

" [**UNDER** *supertable*] "

Element	Description	Restrictions	Syntax
row_type	Name of the ROW type on which this table is based	Must be a named ROW data type registered in the local database	Identifier on page 1231
supertable	Name of the table from which this table inherits its properties	Must already exist as a typed table	Identifier on page 1231

If you use the UNDER clause, the *row_type* must be derived from the ROW type of the *supertable*. A type hierarchy must already exist in which the named ROW type of the new table is a subtype of the named ROW type of the *supertable*.

Jagged rows are any set rows from a table hierarchy in which the number of columns is not fixed among the typed tables within the hierarchy. Some APIs, such as and , do not support queries that return jagged rows.

When you create a typed table, CREATE TABLE cannot specify names for its columns, because the column names were declared when you created the ROW type. Columns of a typed table correspond to the fields of the named ROW type. The ALTER TABLE statement cannot add additional columns to a typed table.

For example, suppose you create a named ROW type, **student_t**, as follows:

```
CREATE ROW TYPE student_t
   (name        VARCHAR(30),
    average     REAL,
    birthdate   DATETIME YEAR TO DAY);
```

If a table is assigned the type **student_t** in the OF TYPE clause, the table is a typed table whose columns are of the same name and data type, and in the same order, as the fields of the named ROW type **student_t**. For example, the following CREATE TABLE statement creates a typed table named **students** whose type is **student_t**:

```
CREATE TABLE students OF TYPE student_t;
```

The **students** table has the following columns:

```
name      VARCHAR(30)
average   REAL
birthdate DATETIME YEAR TO DAY
```

For more information about named ROW types, refer to the CREATE ROW TYPE statement on page 286.

Using Large-Object Data in Typed Tables

Use the BLOB or CLOB instead of BYTE or TEXT data types when you create a typed table that contains columns for large objects. For backward compatibility, you can create a named-ROW type that contains BYTE or TEXT fields and use that ROW type to re-create an existing (untyped) table as a typed table. Although you can use a ROW type that contains BYTE or TEXT fields to create a typed table, such a ROW type is not valid as a column. You can, however, use a ROW type that contains BLOB or CLOB fields both in typed tables and in columns.

Using the UNDER Clause

Use the UNDER clause to specify inheritance (that is, define the new table as a subtable). The subtable inherits properties from the specified supertable. In addition, you can define new properties specific to the subtable.

Continuing the example shown in OF TYPE Clause on page 343, the following statements create a typed table, **grad_students**, that inherits all of the columns of the **students** table but also has columns for **adviser** and **field_of_study** that correspond to fields in the **grad_student_t** ROW type:

```
CREATE ROW TYPE grad_student_t
   (adviser        CHAR(25),
    field_of_study CHAR(40)) UNDER student_t;

CREATE TABLE grad_students OF TYPE grad_student_t  UNDER students;
```

When you use the UNDER clause, the subtable inherits these properties:

- All columns in the supertable
- All constraints defined on the supertable
- All indexes defined on the supertable
- All triggers defined on the supertable.
- All referential integrity constraints
- The access method
- The storage option specification (including fragmentation strategy)

  If a subtable defines no fragments, but if its supertable has fragments defined, then the subtable inherits the fragments of the supertable.

Tip: Any heritable attributes that are added to a supertable after subtables have been created are automatically inherited by existing subtables. You do not need to add all heritable attributes to a supertable before you create its subtables.

Restrictions on Table Hierarchies

Inheritance occurs in one direction only, namely from supertable to subtable. Properties of subtables are *not* inherited by supertables. The section System Catalog Information on page 345 lists the inherited database objects for which the system catalog maintains no information regarding subtables.

No two tables in a table hierarchy can have the same data type. For example, the final line of the next code example is invalid, because the tables **tab2** and **tab3** cannot have the same row type (**rowtype2**):

```
create row type rowtype1 (...);
create row type rowtype2 (...) under rowtype1;
create table tab1 of type rowtype1;
create table tab2 of type rowtype2 under tab1;
create table tab3 of type rowtype2 under tab1; -- This is not valid.
```

Access Privileges on Tables

The discretionary access privileges on a table describe both who can access the information in the table and who can create new tables.

For more information about discretionary access privileges, see the description of the GRANT statement on page 537 statement.

In an ANSI-compliant database, no default table-level privileges exist. You must grant these privileges explicitly.

Setting the environment variable **NODEFDAC** to `yes` prevents default privileges from being granted to PUBLIC on new tables in a database that is not ANSI compliant, as described in the *HCL® Informix® Guide to SQL: Reference*.

Important: Enabling **NODEFDAC** withholds default table access or routine access privileges from PUBLIC when the object is registered in the system catalog, but the **NODEFDAC** setting cannot prevent the PUBLIC group from being granted the same privileges by a user who holds the necessary access privileges on the new table or on the new UDR.

For more information about discretionary access privileges, see the *HCL® Informix® Guide to SQL: Tutorial*.

System Catalog Information

When you create a table, the database server adds information about the table to the **systables** system catalog table, and column information to **syscolumns** system catalog table. The **sysfragments** system catalog table contains information about fragmentation strategies and the storage location of fragments. The **sysblobs** system catalog table contains information about the location of dbspaces and of simple large objects. (The **syschunks** table in the **sysmaster** database contains information about the location of smart large objects.)

The **systabauth**, **syscolauth**, **sysfragauth**, **sysprocauth**, **sysusers**, and **sysxtdtypeauth** tables contain information about the discretionary access privileges that various CREATE TABLE options require.

The **sysextcols**, **sysextdfiles**, and **sysexternal** tables contain additional information about objects that the CREATE EXTERNAL TABLE statement registers in the database.

The **systables**, **sysxtdtypes**, and **sysinherits** system catalog tables provide information about typed tables. For typed-table hierarchies, constraints, indexes, and triggers are recorded in the system catalog for the supertable, but not for subtables that inherit them. Fragmentation information, however, is recorded for both supertables and subtables. For more information about inheritance, refer to the *HCL® Informix® Guide to SQL: Tutorial*.

Specialized columns

You can create specialized columns to support Enterprise Replication, high availability, or row-level auditing.

```
Specialized columns

     .WITH--+-AUDIT-----+-------------------------------><
            '-CRCOLS----'
            '-ERKEY-----'
            '-REPLCHECK-'
            '-VERCOLS---'
```

There is no required order among multiple WITH options in the same CREATE TABLE statement.

Using the WITH AUDIT Clause

Use the WITH AUDIT keywords to create a table that will be included in the set of tables that are audited at the row level if selective row-level is enabled.

When you create a table with the WITH AUDIT clause, row-level audit events in that table are recorded when selective row-level auditing is turned on. Applying the WITH AUDIT attribute to a table by itself does not enable selective row-level auditing. This type of auditing is enabled when the ADTROWS parameter of the `adtcfg` file is set to 1 or 2 by using the onaudit -R command.

Using the WITH CRCOLS Option

Use the WITH CRCOLS keywords to create two shadow columns that uses for conflict resolution. The first column, **cdrserver**, contains the identity of the database server where the last modification occurred. The second column, **cdrtime**, contains the time stamp of the last modification. You must add these columns before you can use time stamps for UDR conflict resolution. These two columns are hidden shadow columns, because they cannot be indexed and cannot be viewed in system catalog tables.

For most database operations, the **cdrserver** and **cdrtime** columns are hidden. For example, if you include the WITH CRCOLS keywords when you create a table, the **cdrserver** and **cdrtime** columns have the following behavior:

- They are not returned by queries that specify an asterisk (*) as the projection list, as in the statement:

  ```
  SELECT * FROM tablename;
  ```

- They do not appear in when you ask for information about the columns of the table.
- They are not included in the number of columns (**ncols**) in the **systables** system catalog table entry for *tablename*.

To view the contents of **cdrserver** and **cdrtime**, you must explicitly specify the columns in the projection list of a SELECT statement, as the following example shows:

```
SELECT cdrserver, cdrtime FROM tablename;
```

For more information about how to use this option, refer to the .

Using the WITH ERKEY Keywords

Use the WITH ERKEY keywords to create the ERKEY shadow columns that uses for a replication key.

The ERKEY shadow columns (**ifx_erkey_1**, **ifx_erkey_2**, and **ifx_erkey_3**) are visible shadow columns because they are indexed and can be viewed in system catalog tables. After you create the ERKEY shadow columns, a new unique index and a unique constraint are created on the table using these columns. Enterprise Replication uses that index as the replication key.

For most database operations, the ERKEY columns are hidden. For example, if you include the WITH ERKEY keywords when you create a table, the ERKEY columns have the following behavior:

- They are not returned by queries that specify an asterisk (*) as the projection list, as in the statement:

  ```
  SELECT * FROM tablename;
  ```

- They do appear in when you ask for information about the columns of the table.
- They are included in the number of columns (**ncols**) in the **systables** system catalog table entry for *tablename*.

To view the contents of the ERKEY columns, you must explicitly specify the columns in the projection list of a SELECT statement, as the following example shows:

```
SELECT ifx_erkey_1, ifx_erkey_2, ifx_erkey_3 FROM customer;
```

Example

Example

In the following example, the ERKEY shadow columns are added to the **customer** table:

```
CREATE TABLE customer (id INT) WITH ERKEY;
```

Using the WITH REPLCHECK Keywords

Use the WITH REPLCHECK keywords to create the **ifx_replcheck** shadow column that uses for consistency checking.

The **ifx_replcheck** column is a visible shadow column because it can be indexed and can be viewed in system catalog tables. After you create the **ifx_replcheck** shadow column, you must create a new unique index on the primary key and the **ifx_replcheck** column. The **ifx_replcheck** shadow column must be the last column in the index. Enterprise Replication uses that index to speed consistency checking.

For most database operations, the **ifx_replcheck** column is hidden. For example, if you include the WITH REPLCHECK keywords when you create a table, the **ifx_replcheck** column has the following behavior:

- It is not returned by queries that specify an asterisk (*) as the projection list, as in the statement:

  ```
  SELECT * FROM tablename;
  ```

- It does appear in when you ask for information about the columns of the table.
- It is included in the number of columns (**ncols**) in the **systables** system catalog table entry for *tablename*.

To view the contents of the **ifx_replcheck** column, you must explicitly specify the columns in the projection list of a SELECT statement, as the following example shows:

```
SELECT ifx_replcheck FROM customer;
```

Example

Example

In the following example, the **ifx_replcheck** shadow column is added to the **customer** table:

```
CREATE TABLE customer (id int) WITH REPLCHECK;
```

Using the WITH VERCOLS Option

Use the WITH VERCOLS keywords to create two shadow columns that Informix® uses to support update operations on secondary servers.

The first column, **ifx_insert_checksum**, contains a checksum of the row when it was first created. The second column, **ifx_row_version**, contains a version number of the row. When a row is first inserted, **ifx_insert_checksum** is generated, and **ifx_row_version** will be set to one. Each time the row is updated, **ifx_row_version** is incremented by one, but **ifx_insert_checksum** does not change. These two columns are visible shadow columns because they can be indexed and can be viewed in system catalog tables.

For most database operations, the **ifx_insert_checksum** and **ifx_row_version** columns are hidden. For example, if you include the WITH VERCOLS keywords when you create a table, the **ifx_insert_checksum** and **ifx_row_version** columns have the following behavior:

- They are not returned by queries that specify an asterisk (*) as the projection list, as in the statement:

  ```
  SELECT * FROM tablename;
  ```

- They appear in when you ask for information about the columns of the table.
- They are included in the number of columns (**ncols**) in the **systables** system catalog table entry for *tablename*.

To view the contents of **ifx_insert_checksum** and **ifx_row_version**, you must explicitly specify the column names in the projection list of a SELECT statement, as the following example shows:

```
SELECT ifx_insert_checksum, ifx_row_version FROM tablename;
```

When row versioning is enabled, **ifx_row_version** is incremented by one each time the row is updated; however, row updates made by Enterprise Replication do not increment the row version. To update the row version on a server using Enterprise Replication, you must include the **ifx_row_version** column in the replicate participant definition.

For more information about how to use this option, refer to the *HCL® Informix® Administrator's Guide*.

SECURITY POLICY Clause

The optional Security Policy clause can use the following syntax to specify the name of an existing security policy that is thereby associated with the table.

This syntax fragment is part of the CREATE TABLE statement on page 310.

SECURITY POLICY Clause

" SECURITY POLICY*policy* "

Element	Description	Restrictions	Syntax
policy	Name of a security policy	Must exist in the database	Identifier on page 1231

Usage

Only DBSECADM can create a table that includes the Security Policy clause to specify a security policy for the table.

Restrictions on adding a security policy

The following guidelines apply to tables that can be protected by including a valid SECURITY POLICY clause in the CREATE TABLE statement, and that also include a column of data type IDSSECURITYLABEL that stores an LBAC label component of the same security policy.

- A table is not protected unless it has a security policy associated with it and has either rows secured, or has at least one column secured.
  - Having rows secured indicates that the table is a *protected table* with *row-level* granularity.
  - Having at least one column secured indicates that the table is a *protected table* with *column-level* granularity.
- Securing rows with the IDSSECURITYLABEL column clause fails if the table does not have a security policy associated with it.
- Securing a column with the COLUMN SECURED WITH clause fails if the table does not have a security policy associated with it.
- A table can have at most one security policy.
- A table can have any number of protected columns. Each protected column can have a different security label, or several protected columns can share the same security label, but all labels must have the same security policy.
- A security policy cannot be associated with a temporary table, nor with a typed table in a table hierarchy.

Storage options

Use the FRAGMENT BY clause, PUT clause, EXTENT size options, and COMPRESSED option of the CREATE TABLE statement to specify the storage location, the distribution scheme, the extent size for the table, and whether the table is enabled for the automatic compression of large amounts of new row data.

This syntax fragment is part of the CREATE TABLE statement on page 310.

(explicit id unique_448_Connect_42_stoop003) unique_448_Connect_42_stoop003(explicit id unique_448_Connect_42_stoop004)
unique_448_Connect_42_stoop004(explicit id unique_448_Connect_42_stoop005) unique_448_Connect_42_stoop005(explicit id
unique_448_Connect_42_stoop006) unique_448_Connect_42_stoop006

Storage Options

" [{ IN { *dbspace* | *extspace* } | <FRAGMENT BY clause> [] }] "

" [<PUT clause> []] "

" [<EXTENT SIZE options> []] "

" [<COMPRESSED option> []] "

Element	Description	Restrictions	Syntax
dbspace	Dbspace to store the table	Must exist. For tables in tenant databases, see **Dbspaces in tenant databases** below.	Identifier on page 1231
extspace	Name declared in the onspaces command to a storage area outside the database server	Must exist	See documentation for your access method.

Usage

The storage options that specify the location, distribution scheme, and extent size for the table are an extension to the ANSI/ISO standard for SQL syntax.

If you use the USING access-method clause to specify an access method, that method must support the storage space.

You can specify a dbspace for the table that is different from the storage location for the database, or fragment the table among dbspaces, or among named fragments in one or more dbspaces.

If you specify no IN clause nor fragmentation scheme, the new table is stored in the same dbspace where the current database is stored. However, if you enabled automatic location and fragmentation, tables are created and fragmented in dbspaces that are chosen by the server. To enable the automatic location and fragmentation of tables, set the AUTOLOCATE

configuration parameter or session environment variable to a positive integer. The value of the integer represents the number of fragments to initially allocate to the table. Additional fragments are added as the table grows.

Sbspaces for smart large objects

You can use the PUT clause to specify sbspace storage locations and storage characteristics for smart large objects, such as BLOB or CLOB column values.

Note: If your table contains simple large objects (TEXT or BYTE), you can specify a separate blobspace for each object.

Dbspaces in tenant databases

If the table is in a tenant database, the *dbspace* must be a dedicated dbspace in the tenant database properties list. If the table is not in a tenant database, the *dbspace* cannot be the name of a dbspace that is dedicated to a tenant database.

Using the IN Clause

Use the IN clause of the CREATE TABLE statement to specify a storage space for the table. The storage space that you specify must already exist.

You can use the IN clause to isolate a table. For example, if the **history** database is in the **dbs1** dbspace, but you want the **family** data placed in a separate dbspace called **famdata**, use the following statements:

```
CREATE DATABASE history IN dbs1;

CREATE TABLE family
   (
   id_num      SERIAL(101) UNIQUE,
   name        CHAR(40),
   nickname    CHAR(20),
   mother      CHAR(40),
   father      CHAR(40)
   )
   IN famdata;
```

For more information about how to store and manage your tables in separate dbspaces, see your *HCL® Informix® Administrator's Guide*.

In general, use the *extspace* storage option in conjunction with the USING Access-Method Clause on page 367. For more information, refer to the documentation of your access method.

If IN *dbspace* is the only storage specification for the new table, then by default 16 kilobytes of storage (or enough storage for four pages, if 4 pages require more than 16 kilobytes) are allocated for the first extent size at the time of table creation.

No storage is allocated for the first extent, however, if the CREATE TABLE statement includes none of the following storage specifications:

- EXTENT SIZE
- NEXT SIZE
- IN *dbspace*.

In this case, storage allocation for the first extent is deferred until the first row is stored in the table.

The same storage allocation deferral applies to tables defined by the CREATE TEMP TABLE statement that do not include any of the storage specifications listed above.

When rows are first inserted into a table for which extent allocation has been deferred, the default size for the first extent is 16 kilobytes. If 16 kilobytes are insufficient for 4 pages, the first extent size will be 4 pages.

When reverting a Version 11.70 or later database server to an earlier release that does not support deferred extent allocation for empty tables, you must either drop or else insert a row into any empty database table that you wish to revert. Empty external tables that do not support INSERT operations must be dropped.

PUT Clause

Use the PUT clause to specify the storage spaces and their characteristics for each column that will contain smart large objects.

This syntax fragment is part of the Storage options on page 350.

PUT Clause

```
"PUT"

"{| column IN ( sbspace ) | ( [{| EXTENT SIZE kilobytes | [{ NO LOG | LOG }] | [{ HIGH INTEG | MODERATE INTEG }] |
[{ NO KEEP ACCESS TIME | KEEP ACCESS TIME }] ] ) }"
```

Element	Description	Restrictions	Syntax
column	Column to store in *sbspace*	Must contain a BLOB, CLOB, user-defined, or complex data type	Identifier on page 1231
kilobytes	Number of kilobytes to allocate for the extent size	Must be an integer value	Literal Number on page 1198
sbspace	Name of a storage area for smart large objects	Must exist If the table is in a tenant database, the *sbspace* must be a dedicated sbspace in the tenant database properties list. If the table is not in a tenant database, the *sbspace* cannot be the name of an	Identifier on page 1231

Element	Description	Restrictions	Syntax
		sbspace that is dedicated to a tenant database.	

The specified column cannot be in the form *column.field*. That is, the smart large object that you are storing cannot be one field of a ROW type.

Specifying the storage location

Each smart large object is stored in a single sbspace. The SBSPACENAME configuration parameter specifies the system default sbspace in which smart large objects are created, unless the PUT clause specifies another sbspace.

For example, the following statement defines **tabwblob** as a table whose only column is of data type BLOB. The column name is declared as **image01**, and the PUT clause specifies the storage location for all of its BLOB objects as **sbspace01**:

```
CREATE TABLE tabwblob
   (
    image01 BLOB
   ) PUT image01 IN (sbspace01);
```

For the example above to be valid, the **sbspace01** must already exist. Because the no other options to the PUT clause are specified, **sbspace01** has default values for its extent size and for the other storage characteristics that the PUT clause can define, including `NO LOG`, `HIGH INTEG`, and `NO KEEP ACCESS TIME`, as defined below.

The PUT clause can specify storage locations for a list of BLOB and CLOB columns. The following example defines **tabw2blobs** as a table with two columns, where column **image02** is of type BLOB, and column **commentary03** is of type CLOB. In the next example, the PUT clause specifies that all the smart large objects in both columns are stored in the same **sbspace01** smart large object space:

```
CREATE TABLE tabw2blobs
   (
    image02 BLOB,
    commentary03 CLOB
   ) PUT image02 IN (sbspace01),
       commentary03 IN (sbspace01);
```

You can specify that more than one sbspace stores the same BLOB or CLOB column. This distributes the smart large objects in a round-robin distribution scheme, so that the number of smart large objects in each sbspace is approximately equal. The comma-separated list of sbspaces for a single column must be delimited by parentheses.

The next example defines **tabw2sblobs** as a table with two columns, where column **image04** is of type BLOB, and column **commentary05** is of type CLOB. The PUT clause specifies that the BLOB objects in column **image04** are stored in two sbspaces, **sbspace01** and **sbspace02**, and all the CLOB objects in column **image05** are stored in sbspace **sbspace03**:

```
CREATE TABLE tabw2sblobs
   (
    image04 BLOB,
    commentary05 CLOB
   ) PUT image04 IN (sbspace01,sbspace02),
       commentary05 IN (sbspace03);
```

If an INSERT or MERGE operation adds six new rows to the table in this example,

- three of the **image04** BLOB objects of will be stored in **sbspace01**,
- the other three **image04** BLOB objects will be stored in **sbspace02**,
- and all six **commentary05** CLOB objects will be stored in **sbspace03**.

When you distribute smart large objects across different sbspaces, you can work with smaller sbspaces. If you limit the size of an sbspace, backup and archive operations can perform more quickly.

Specifying sbspace characteristics

The following storage options are available to store BLOB and CLOB data:

Option

Effect

EXTENT SIZE

Specifies a lower limit on how many kilobytes can be stored in a smart-large-object extent. The database server might round the specified *kilobytes* value up, so that the extent size is an integer multiple of the sbspace page size.

HIGH INTEG

This high data-integrity option produces user-data pages that contain a page header and a page trailer to detect incomplete writes and data corruption. This option is the default data-integrity behavior.

MODERATE INTEG

This data-integrity option produces user-data pages that contain a page header but no page trailer. This option cannot compare the page header with the page trailer to detect incomplete writes and data corruption.

KEEP ACCESS TIME

This maintains a record in the smart-large-object metadata of the system time when the smart large object was last read or written.

NO KEEP ACCESS TIME

Does not record the system time when the smart large object was last read or written. This provides better performance than the KEEP ACCESS TIME option, and is the default tracking behavior.

LOG

Follows the logging procedure used with the current database log for the corresponding smart large object. This option can generate large amounts of log traffic and increase the risk of filling the logical log.

NO LOG

Turns off logging. This option is the default behavior.

The comma-separated list of keyword options that define sbspace characteristics must be enclosed in parentheses, and immediately follows the sbspace (or the list of sbspaces) that stores the BLOB or CLOB column. In the following

example, the PUT clause specifies that the unlogged **sbspace01** and **sbspace02** sbspaces that store the BLOB objects of column **image04** have characteristics different from **sbspace03**, a logged sbspace that stores CLOB objects of column **commentary05**:

```
CREATE TABLE tabw2sblobs
   (
    image04 BLOB,
    commentary05 CLOB
   ) PUT image04 IN (sbspace01,sbspace02) (KEEP ACCESS TIME, MODERATE INTEG),
       commentary05 IN (sbspace03) (EXTENT SIZE 30, LOG);
```

When you turn logging on for a smart large object, you must immediately perform a level-0 backup to be able to recover and restore the smart large object.

The **syscolattribs** system catalog table contains one row for each *sbspace* and *column* combination in the PUT clause:

- The **syscolattribs.extentsize** column stores the extent size, based on the *kilobytes* value.
- The **syscolattribs.flags** column stores a bitmap corresponding to the logging and access time status, and data integrity setting.

If a user-defined or complex data type contains more than one large object, the specified large-object storage options apply to all large objects in the type, unless the storage options are overridden when the large object is created.

Important: The PUT clause does not affect the storage of simple-large-object data types (BYTE and TEXT). For information on how to store BYTE and TEXT data, see Large-Object Data Types on page 993.

Alternative to full logging

Instead of full logging, you can turn off logging when you load the smart large object initially and then turn logging back on once the object is loaded.

Use the NO LOG option to turn off logging. If you use NO LOG, you can restore the smart-large-object metadata later to a state in which no structural inconsistencies exist. In most cases, no transaction inconsistencies will exist either, but that result is not guaranteed.

The following statement creates the **greek** table. Data values for the table are fragmented into the **dbs1** and **dbs2** dbspaces. The PUT clause assigns the smart-large-object data in the **gamma** and **delta** columns to the **sb1** and **sb2** sbspaces, respectively. The TEXT data values in the **eps** column are assigned to the **blb1** blobspace.

```
CREATE TABLE greek
(alpha INTEGER,
 beta  VARCHAR(150),
 gamma CLOB,
 delta BLOB,
 eps   TEXT IN blb1)
   FRAGMENT BY EXPRESSION
   alpha <= 5 IN dbs1, alpha > 5 IN dbs2
   PUT gamma IN (sb1), delta IN (sb2);
```

FRAGMENT BY clause

Use the FRAGMENT BY clause to create a fragmented table and to specify its storage distribution scheme. The keywords PARTITION BY are a synonym for FRAGMENT BY.

This syntax fragment is part of the Storage options on page 350.

(explicit id unique_77_Connect_42_fbcl005) unique_77_Connect_42_fbcl005(explicit id unique_77_Connect_42_fbcl006) unique_77_Connect_42_fbcl006(explicit id unique_77_Connect_42_fbcl007) unique_77_Connect_42_fbcl007

FRAGMENT BY clause for tables

" [**WITH ROWIDS**] "

" { **FRAGMENT** | **PARTITION** } **BY** "

" { **ROUND ROBIN** { **IN** *dbspace* | **PARTITION** *partition* **IN** *dbspace* } | **EXPRESSION** <Expression Fragment Clause> [] | **RANGE** (*fragment_key*) <Interval Fragment Clause> [] | **LIST** (*fragment_key*) <List Fragment Clause> [] } "

Element	Description	Restrictions	Syntax
column	Column to which to apply the fragmentation strategy	Must be a column within the table	Identifier on page 1231
dbspace	Dbspace to store the table fragment	You can specify no more than 2,048 *dbspaces*. All *dbspaces* that store the fragments must have the same page size.	Identifier on page 1231
fragment_key	Cast, column, or function expression on a table column. This is the expression on which the table is fragmented.	Columns must be from the current table only	Expression on page 1003
partition	Name declared here for a fragment	Must be unique among the names of fragments of the table	Identifier on page 1231

When you fragment a table, the IN keyword is followed by the name of the storage space where a table fragment is to be stored.

Using the WITH ROWIDS Option

Nonfragmented tables contain a hidden column called **rowid**, but by default, fragmented tables have no **rowid** column. You can use the WITH ROWIDS keywords to add the **rowid** column to a fragmented table. Each row is automatically assigned a unique **rowid** value that remains stable for the life of the row and that the database server can use to find the physical location of the row. Each row requires an additional four bytes to store the **rowid**.

Important:

This is a deprecated feature. The query optimizer might not use an index scan when explicit **rowid** shadow columns are defined on fragmented tables. When creating new applications, use primary keys as a method of row identification instead of using **rowid** values.

You cannot use the WITH ROWIDS clause with typed tables.

Fragmenting by ROUND ROBIN

In a round-robin distribution scheme, specify at least two dbspaces where you want the fragments to be placed, or specify at least two fragment names in one or more dbspaces. As records are inserted into the table, they are placed in the first available fragment. The database server balances the load among the specified fragments as you insert records and distributes the rows in such a way that the fragments always maintain approximately the same number of rows (±100). A newly-added round-robin fragment will be favored exclusively by inserts and loads until it no longer has the fewest rows among the table's fragments.

In this distribution scheme, the database server must scan all fragments when it searches for a row.

Important:

The FRAGMENT BY ROUND ROBIN clause overrides the automatic location and fragmentation of tables, which is enabled when the AUTOLOCATE configuration parameter or the AUTOLOCATE session environment option is set to a positive integer.

When automatic location and fragmentation is enabled, the database server automatically determines

- the table extent sizes,
- the dbspaces where the fragments are stored,
- and a ROUND ROBIN distributed-storage strategy for the new table.

Round-robin fragmentation with large object data types

For simple large objects in tables that contain BYTE or TEXT columns and that are fragmented by round-robin, you can reserve space for inserting BYTE and TEXT data by setting the PN_STAGEBLOB_THRESHOLD configuration parameter. For information about how the database server stages simple large objects in round-robin fragments, see the description of PN_STAGEBLOB_THRESHOLD in your *HCL® Informix® Administrator's Reference*.

For smart large objects in tables that contain BLOB or CLOB columns, you can use the PUT clause to specify round-robin fragmentation in a list of sbspaces. When you include the PUT clause in the CREATE TABLE statement (or in the CREATE TEMP TABLE statement or in the ALTER TABLE statement), you have the options to include or not include a FRAGMENT BY clause defining distributed storage for other columns in the same table. The PUT clause applies a round-robin storage

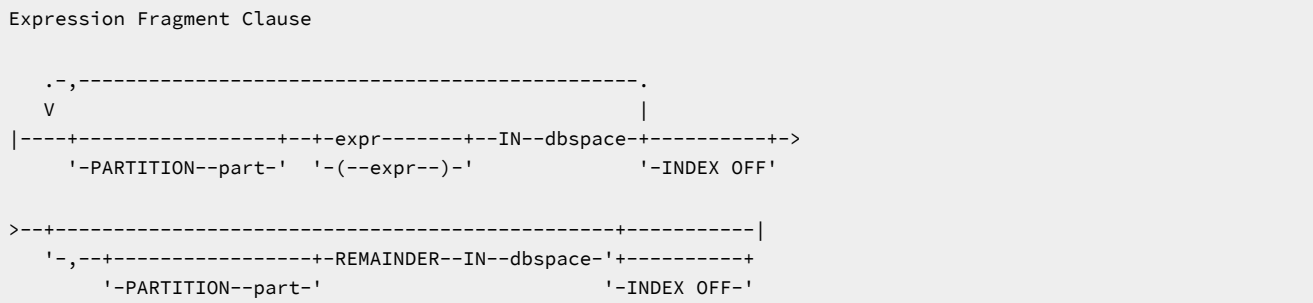
distribution strategy only to smart large object columns for which it specifies more than one sbspace. For more information and examples, see the PUT Clause on page 352.

Expression Fragment Clause

In an *expression-based* distribution scheme, each fragment expression in a rule specifies a storage space. Each fragment expression in the rule isolates data and aids the database server in searching for rows.

This syntax fragment is part of the FRAGMENT BY clause on page 356.

Expression Fragment Clause



Element	Description	Restrictions	Syntax
part	Name of a fragment	Required if *part* is stored in the same dbspace as another fragment of this table. Must be unique among names of fragments of the same table.	Identifier on page 1231
dbspace	dbspace to store the table fragment	You can specify no more than 2,048 *dbspaces*. All *dbspaces* that store the fragments must have the same page size.	Identifier on page 1231
dbspace			
expr	An expression, based on column values, defining a fragment	Must return a Boolean value (true or false). Data values must be from a single row of the table.	Expression on page 1003

To fragment a table by expression, specify one of the following rules:

- Range rule

  A range rule specifies fragment expressions that use a range to specify which rows are placed in a fragment, as the next example shows:

  ```
  FRAGMENT BY EXPRESSION c1 < 100 IN dbsp1,
    c1 >= 100 AND c1 < 200 IN dbsp2, c1 >= 200 IN dbsp3;
  ```

- Arbitrary rule

An arbitrary rule specifies fragment expressions based on a predefined SQL expression that typically uses OR clauses to group data, as the following example shows:

```
FRAGMENT BY EXPRESSION
   zip_num = 95228 OR zip_num = 95443 IN dbsp2,
   zip_num = 91120 OR zip_num = 92310 IN dbsp4,
   REMAINDER IN dbsp5;
```

Warning: See the note about the **DBCENTURY** environment variable and date values in fragment expressions in the section Logging Options on page 319.

REMAINDER keyword

Use the REMAINDER keyword to specify the storage space in which to store valid values that fall outside the specified expression or expressions. If you do not specify a remainder, and a row is inserted or updated with values that do not correspond to any fragment definition, the database server returns an error.

The following example uses an arbitrary rule to define five fragments for specific values of the **c1** column, and a sixth fragment for all other values:

```
CREATE TABLE T1 (c1 INT) FRAGMENT BY EXPRESSION
   PARTITION PART_1 (c1 = 10) IN dbs1,
   PARTITION PART_2 (c1 = 20) IN dbs1,
   PARTITION PART_3 (c1 = 30) IN dbs1,
   PARTITION PART_4 (c1 = 40) IN dbs2,
   PARTITION PART_5 (c1 = 50) IN dbs2,
   PARTITION PART_6 REMAINDER IN dbs2;
```

Here the first three fragments are stored in partitions of the **dbs1** dbspace, and the other fragments, including the remainder, are stored in named fragments of the **dbs2** dbspace. Explicit fragment names are required in this example, because each dbspace has multiple partitions.

Fragmentation in NLCASE INSENSITIVE databases

In databases with the NLSCASE INSENSITIVE property, operations on NCHAR and NVARCHAR data ignore lettercase, so that the database server treats case variants among strings composed of same sequence letters as duplicates. If the fragment keys for a table that is fragmented by expression are NCHAR or NVARCHAR columns, then each fragment defined by a character expression stores all lettercase variants that match the expression that defines the fragment. For example, for the expression `lname = 'Garcia'` where **lname** is a column of type NCHAR or NVARCHAR, rows with the following values in that column would all be stored in the same fragment, because the case-insensitive expression evaluates to TRUE for these (and similar) string values:

```
'Garcia' 'garcia' 'GARCIA' 'GarCia' 'gARCia'
```

For more information about NLSCASE INSENSITIVE databases, see CREATE DATABASE statement on page 190, Duplicate rows in NLSCASE INSENSITIVE databases on page 699, and NCHAR and NVARCHAR expressions in case-insensitive databases.

User-Defined Functions in Fragment Expressions

For rows that include user-defined data types, you can use comparison conditions or user-defined functions to define the range rules. In the following example, comparison conditions define the range rules for the **long1** column, which contains an opaque data type:

```
FRAGMENT BY EXPRESSION
   long1 < '3001' IN dbsp1,
   long1 BETWEEN '3001' AND '6000' IN dbsp2,
   long1 > '6000' IN dbsp3;
```

An implicit, user-defined cast converts `3001` and `6000` to the opaque type.

Alternatively, you can use user-defined functions to define the range rules for the opaque data type of the **long1** column:

```
FRAGMENT BY EXPRESSION
   (lessthan(long1,'3001')) IN dbsp1,
   (greaterthanorequal(long1,'3001') AND
   lessthanorequal(long1,'6000')) IN dbsp2,
   (greaterthan(long1,'6000')) IN dbsp3;
```

Explicit user-defined functions require parentheses around the entire fragment expression before the IN clause, as the previous example shows.

User-defined functions in a fragment expression can be written in SPL or in the C or Java™ language. These functions must satisfy four requirements:

- They must evaluate to a Boolean value.
- They must be nonvariant.
- They must reside within the same database as the table.
- They must not generate OUT nor INOUT parameters.

For information on how to create UDRs for fragment expressions, refer to *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

Partial Index in Fragment Expressions

Starting 14.10xC2 onwards, Informix supports FRAGMENT BY EXPRESSION partial index.

A *partial index* is an index built over a subset of a table. The subset is defined by a conditional expression (called the *predicate* of the partial index). The index contains entries for only those table rows that satisfy the predicate.

Partial indexes avoid indexing common values. As the query searching for a common value will not use the index, there is no point in keeping those rows in the index. This reduces the size of the index, which will speed up queries that do use the index. It will also speed up many table update operations because the index does not need to be updated in all cases.

Example 1: Setting up a Partial Index to Exclude Common Values

```
create table tab1 (n int, n2 int, n4 int, cc char(16));
-- insert 10,000 rows
with cte(n) as (
```

```
 select 1 as n
 union all
 select n+1 from cte where n < 10000
 )
insert into tab1
  select n,
   case when mod(n,10) == 0 then n else null end,
   mod(n,4),
   n from cte;
```

To create a partial index that suits the above example for non-null values, create index as:

```
create index idx1 on tab1(n) fragment by expression
        (n2 is null) in dbs1 INDEX OFF,
 remainder in dbs2;

“oncheck -pk | grep ‘^Key’ | wc -l” shows only 1000 rows are indexed.
```

The query that can use this index would be:

```
select count(*) from tab1 where n < 1000 and n2 is not null;
```

The query that cannot use this index would be:

```
select count(*) from tab1 where n < 1000;
```

Example 2: Setting up a Partial Index to Exclude Un-wanted Values

If you have a table that contains customers from different states, where one specific state records take up a small fraction of the total table and yet those are the most-accessed rows, you can improve performance by creating an index on just that specific state rows. The command to create the index:

```
create index zip_ix on customer(zipcode)
    fragment by expression
        (state = 'CA') in dbs1,
        remainder in INDEX OFF;
```

The query that can use this index would be:

```
select fname, lname from customer
        where zipcode >= '94117'
        and state='CA';
```

Note: A partial index can be used in a query only if the system can recognize that the WHERE condition of the query mathematically implies the predicate of the index..

List fragment clause

Use the List Fragment clause to specify a list of fragment key values to store in the same fragment. The rows assigned to each fragment must match the fragment key value (or one of a comma-separated list of fragment key values) that defines the fragment.

This syntax fragment is part of the FRAGMENT BY clause on page 356.

(explicit id unique_88_Connect_42_mod0020203) unique_88_Connect_42_mod0020203

List Fragment Clause

" **PARTITION***partition* <List Expression Clause>**IN***dbspace* "

" [[] , **PARTITION***partition* **REMAINDERIN***dbspace*] "

List Expression Clause

" **VALUES** { (*const_expr*) | [] **[IS] NULL** } "

Element	Description	Restrictions	Syntax
const_expr	Constant expression that defines the list of fragment key values for the fragment to store	Must be a quoted string or a literal value. Each value in the list must be unique among the lists for fragments of the same object.	Constant Expressions on page 1040
dbspace	dbspace to store the fragment	You can specify no more than 2,048 *dbspaces*. All of these *dbspaces* must have the same page size.	Identifier on page 1231
partition	Name that you declare here for a fragment	Must be unique among the names of fragments of the same object. If a table and its index use the same list fragmentation strategy, each index fragment must have the same name as the corresponding table fragment.	Identifier on page 1231

Fragmenting by list defines fragments that are each based upon a list of discrete values of the fragment key.

You can use this fragmentation strategy when the values of the fragment key are categories on a nominal scale that has no quantified order within the set of categories. Fragmenting by list is useful when a table contains a finite set of values for the fragment key and queries on the table have an equality predicate on the fragment key. For example, you can fragment data geographically, based on a list of the states or provinces within a country. The rows that are stored in each fragment can be restricted to a single fragment key value, or to a list of values representing some logical subset of fragment key values, provided that no fragment key value is shared by two or more fragments.

Fragmenting by list also helps to logically segregate data.

Fragmenting by list supports these features:

- Both a table and its indexes can be fragmented by list.
- The fragment key can be a column expression based on a single column or on multiple columns.

- The list can optionally include a remainder fragment.
- The list can optionally include a NULL fragment that stores only NULL values.

Fragmenting a table by list (or fragmenting an index, in the CREATE INDEX statement) must satisfy these requirements:

- The list that includes NULL (or IS NULL) cannot include any other value.
- The fragment key must be based on a single row.
- The fragment key must be a column expression. This *constant expression* can be based on a single column or on multiple columns.
- Lists cannot include duplicate *constant expression* values. Each value must be unique within the FRAGMENT BY LIST clause.

Load, INSERT, MERGE, or UPDATE operations on tables fragmented BY LIST can fail at runtime under these circumstances:

- The fragment key for a row evaluates to NULL, but the FRAGMENT BY LIST clause defined no NULL fragment.
- The fragment key for a row matches no *constant expression* value for any fragment, but no remainder fragment is defined.

REMAINDER and NULL fragments in list-based storage distribution

You can optionally define a REMAINDER fragment to store rows that do not match the list of fragment key values for any fragment.

You can optionally define a NULL fragment to store rows with missing fragment key values by specifying only `IS NULL` or `NULL` after the VALUES keyword in the List Expression clause for the fragment. You cannot include `NULL` or `IS NULL` in an expression list that also includes any other expression. (In this context, `NULL` and `IS NULL` are keyword synonyms.)

If no NULL fragment is defined, and an operation attempts to insert a row that is missing data for the fragment key, the result depends on whether a REMAINDER fragment exists:

- If a REMAINDER fragment is defined, the row is stored in the REMAINDER fragment.
- If no REMAINDER fragment is defined, the database server issues an exception.

If no REMAINDER fragment is defined, and an INSERT, UPDATE, MERGE, or other DML operation attempts to store a row whose fragment key does not match a list value for any fragment, the database server issues an exception.

When you define a list-based partitioning scheme for a table or index, the fragment list can include no more than one NULL fragment, and no more than one REMAINDER fragment.

If a table that is partitioned BY LIST has no NULL or REMAINDER fragment, but you subsequently determine that either or both of these fragments are needed, you can add a NULL fragment or a REMAINDER fragment, or both, to the fragment list by using the ADD option to the ALTER FRAGMENT statement. For more information, see ADD Clause on page 47.

EXTENT SIZE Options

The EXTENT SIZE options can define the size of storage extents allocated to the table.

This syntax fragment is part of the Storage options on page 350.

EXTENT SIZE Options

" [**EXTENT SIZE** *first_kilobytes*] "

" [**NEXT SIZE** *next_kilobytes*] "

Element	Description	Restrictions	Syntax
first_kilobytes	Length in kilobytes of the first extent for the table; default is 16 kilobytes.	Must return a positive number; maximum is the chunk size	Expression on page 1003
next_kilobytes	Length in kilobytes of each subsequent extent; default is 16 kilobytes.	Must return a positive number; maximum is the chunk size	Expression on page 1003

Usage

The minimum length of *first_kilobytes* (and of *next_kilobytes*) is four times the disk-page size on your system. For example, if you have a 2-kilobyte page system, the minimum length is 8 kilobytes.

If the CREATE TABLE (or the CREATE TEMP TABLE) statement includes no IN dbspace clause, no EXTENT SIZE specification, and no NEXT SIZE specification, no storage is allocated for the table until at least one data row is inserted into it. The default size of the first extent is either 16 kilobytes or 4 pages.

The next example specifies a first extent of 20 kilobytes and allows the rest of the extents to use the default size:

```
CREATE TABLE emp_info
   (
   f_name     CHAR(20),
   l_name     CHAR(20),
   position   CHAR(20),
   start_date DATETIME YEAR TO DAY,
   comments   VARCHAR(255)
   )
EXTENT SIZE 20;
```

If a table contains no data, you can use the ALTER TABLE MODIFY EXTENT SIZE or ALTER TABLE MODIFY NEXT SIZE statements of SQL to change the size of the first extent and of the next extent of the empty table. These operations are not supported, however, for tables that contain one or more rows. For more information about these options to the ALTER TABLE statement, see #unique_502 and #unique_146.

If you need to revise the extent sizes of a table, you can modify the first extent and next-extent sizes in the generated schema files of an unloaded table. For example, to make a database more efficient, you might unload a table, modify the extent sizes in the schema files, and then create and load a new table. For information about how to optimize extents, see your *HCL® Informix® Administrator's Guide*.

COMPRESSED option for tables

Use the COMPRESSED option of the CREATE TABLE statement to enable the automatic compression of large amounts of row data when the data is loaded into the table or table fragment.

After you create a table with the COMPRESSED option, the database server automatically creates a compression dictionary and compresses the in-row data after 2000 or more rows of data are loaded into a table or fragment. If the data is loaded by a light append, the first 2000 rows and all subsequent rows are compressed. If the data is loaded by another method, all subsequent rows after the first 2000 rows are compressed. To compress the original 2000 rows, run the SQL administration API task() or admin() function with the table compress or fragment compress argument.

The COMPRESSED option enables only the compression of in-row data. The COMPRESSED option does not enable the automatic compression of simple large objects in dbspaces or indexes. (You can use the COMPRESSED keyword of the CREATE INDEX statement to create a compressed B-tree index.)

The following example creates a table that is set up for the automatic compression:

```
CREATE TABLE cust5 (  ...) COMPRESSED;
```

The following example also creates a table named **t**, which defines first and next extent sizes and is set up for the automatic compression:

```
CREATE TABLE t(c int, d int) EXTENT SIZE 32 NEXT SIZE 32 COMPRESSED;
```

To disable the automatic compression of data in new rows, run an SQL administration API task() or admin() function with the table uncompress argument. To re-enable automatic compression, run the SQL administration API task() or admin() function with the table compress argument on the table. You can control compression for a table fragment with the fragment uncompress and fragment compress arguments.

LOCK MODE Options

Use the LOCK MODE options to specify the locking granularity of the new table.

The following table describes the locking-granularity options.

Granularity	Effect
PAGE	Obtains and releases one lock on a whole page of rows If neither the **IFX_DEF_TABLE_LOCKMODE** environment variable nor the **DEF_TABLE_LOCKMODE** configuration parameter has set a default locking granularity, this is the system default. Page-level locking is especially useful when you know that the rows are grouped into pages in the same order that you are using to process all the rows. For example, if you are processing the contents of a table in the same order as its cluster index, page locking is appropriate.
ROW	Obtains and releases one lock per row Row-level locking provides the highest level of concurrency. If you are using many rows at one time, however, the lock-management overhead can become significant. You might also exceed

Granularity	Effect
	the maximum number of locks available, depending on the configuration of your database server, but Informix® can support up to 18 million locks on 32-bit platforms, or 600 million locks on 64-bit platforms. Only tables with row-level locking can support the LAST COMMITTED isolation level feature.

Examples of setting the locking granularity

The following CREATE TABLE statement defines a new table **tsTab_j** with column **id** of type INT and column **ts** of the **TIMESERIES(ts_data_j)** extended data type, with ROW as the locking granularity:

```
CREATE TABLE IF NOT EXISTS tsTab_j(
   id  INT NOT NULL PRIMARY KEY,
   ts  TIMESERIES(ts_data_j)
) LOCK MODE ROW;
```

In the next example, a user who holds the DBSECADAM role creates a **Tab5** table with both row-level and column-level granularity of LBAC protection, and with PAGE as the locking granularity. is

```
CREATE TABLE Tab5 (C1 IDSSECURITYLABEL,
   C2 int,
   C3 char (10) COLUMN SECURED WITH label6)
   SECURITY POLICY company
   LOCK MODE PAGE;
```

This ALTER TABLE . . . LOCK MODE statement can change the locking granularity, as in this example for the **Tab5** table:

```
ALTER TABLE Tab5 LOCK MODE(ROW);
```

Important:

The SET LOCK MODE statement of SQL has no effect on the locking granularity of tables. For the syntax and semantics, see SET LOCK MODE statement on page 817.

Precedence and Default Behavior

In Informix®, you do not need to specify the lock mode each time you create a new table. You can globally set the locking granularity of all new tables in the following environments:

- Database session of an individual user

  You can set the **IFX_DEF_TABLE_LOCKMODE** environment variable to specify the lock mode of new tables during your current session.
- Database server (all sessions on the database server)

If you are a DBA, you can set the DEF_TABLE_LOCKMODE configuration parameter in the ONCONFIG file to determine the lock mode of all new tables in the database server.

If you are not a DBA, you can set the **IFX_DEF_TABLE_LOCKMODE** environment variable for the database server, before you run **oninit**, to specify the lock mode of all new tables of the database server.

The LOCK MODE setting in a CREATE TABLE statement takes precedence over the settings of the **IFX_DEF_TABLE_LOCKMODE** environment variable and the DEF_TABLE_LOCKMODE configuration parameter.

If CREATE TABLE specifies no lock mode setting, the default mode depends on the setting of the **IFX_DEF_TABLE_LOCKMODE** environment variable or the DEF_TABLE_LOCKMODE configuration parameter. For information about **IFX_DEF_TABLE_LOCKMODE**, refer to the *HCL® Informix® Guide to SQL: Reference*. For information about the DEF_TABLE_LOCKMODE configuration parameter, refer to the *HCL® Informix® Administrator's Reference*.

USING Access-Method Clause

The USING Access Method clause can specify an access method.

This syntax fragment is part of the CREATE TABLE statement on page 310.

(explicit id unique_450_Connect_42_uamcl002) unique_450_Connect_42_uamcl002

USING Access-Method Clause

" **USING** "

" <Specific Name> [] "

" [(*config_keyword* [= *'config_value'*])] "

Element	Description	Restrictions	Syntax
config_keyword	Configuration keyword associated with the specified access method	No more than 18 bytes. The access method must exist.	Literal keyword
config_value	Value of the specified configuration keyword	No more than 236 bytes. Must be defined by the access method.	Quoted String on page 1202

A *primary-access method* is a set of routines to perform DDL and DML operations, such as create, drop, insert, delete, update, and scan, to make a table available to the database server. Informix® provides a built-in primary-access method.

You store and manage a virtual table either outside of the database server in an extspace or inside the database server in an sbspace. (See Storage options on page 350.) You can access a virtual table with SQL statements. Access to a virtual table requires a user-defined primary-access method.

modules can provide other primary-access methods to access virtual tables. When you access a virtual table, the database server calls the routines associated with that access method rather than the built-in table routines. For more information on these other primary-access methods, refer to your access-method documentation.

You can retrieve a list of configuration values for an access method from a table descriptor (**mi_am_table_desc**) using the MI_TAB_AMPARAM macro. Not all keywords require configuration values.

The access method must already exist. For example, if an access method called **textfile** exists, you can specify it with the following syntax:

```
CREATE TABLE mybook
   (... )
   IN myextspace
   USING textfile (DELIMITER=':');
```

Statistics options of the CREATE TABLE statement

Use the Statistics Options clause of the CREATE TABLE statement to set the values of the STATCHANGE property of a fragmented or nonfragmented table, and the STATLEVEL property of a fragmented table.

This syntax fragment is part of the CREATE TABLE statement on page 310.

Statistics Options

" { **STATCHANGE AUTO** | [**STATCHANGE** { **AUTO** | *change_threshold* }] } "

" { **STATLEVEL AUTO** | [**STATLEVEL** { **FRAGMENT** | **TABLE** | **AUTO** }] } "

Element	Description	Restrictions	Syntax
change_threshold	Percentage of changed data rows that defines stale distribution statistics	Must be an integer in the range 0 - 100	Literal Number on page 1198

Usage

These table attributes control the threshold for automatic recalculation (STATCHANGE) and the granularity (STATLEVEL) of data distribution statistics.

The Statistics Options clause of the CREATE TABLE statement can define table statistics properties that allow the user to control the actions of

- UPDATE STATISTICS, when that SQL statement is run without the `FOR` keyword,
- and of UPDATE STATISTICS FOR TABLE, when that statement runs in LOW, MEDIUM, or HIGH mode.

The two table properties that the Statistics Options clause can set are STATCHANGE and STATLEVEL.

The STATCHANGE property

The STATCHANGE table attribute specifies the minimum percentage of changes (from UPDATE, DELETE, MERGE, and INSERT operations on the rows in the table or fragment since the previous calculation of distribution statistics) to consider the statistics stale. You can specify an integer value in the range 0 - 100, or you can use the AUTO keyword to apply the current STATCHANGE configuration parameter setting in the ONCONFIG file or in the session environment variable as the default change threshold value.

The automatic mode for selectively updating table and fragment statistics can be enabled in any of the following ways:

- The AUTO_STAT_MODE configuration parameter is set to 1 (or is not set). Enables automatic mode as the system default.
- The AUTO_STAT_MODE session environment variable is set to `ON`. Enables automatic mode during the current session.
- The UPDATE STATISTICS statement includes the `AUTO` keyword. Enables automatic mode while that UPDATE STATISTICS statement is running.

Important:

Enabling automatic mode has no effect, however, on any of these UPDATE STATISTICS statements:

- UPDATE STATISTICS statements that end with the `FORCE` keyword.
- UPDATE STATISTICS statements that include the `FOR FUNCTION`, `FOR PROCEDURE`, `FOR ROUTINE`, or `FOR SPECIFIC` keywords.

While automatic mode is enabled, UPDATE STATISTICS statements use the explicit or default STATCHANGE value to identify table, index, or fragment distributions statistics in the system catalog that are missing or stale, and selectively updates only the missing or stale statistics. For more information about the automatic mode for UPDATE STATISTICS operations, see the description of the AUTO_STAT_MODE configuration parameter in the *HCL® Informix® Administrator's Reference*. See also AUTO_STAT_MODE session environment option and Using the FORCE and AUTO keywords on page 890.

The STATLEVEL property

The STATLEVEL property of a fragmented table can determine the level of granularity of its data distributions and index statistics. It can take one of the following three values, with `AUTO` being the default, if no value is specified at creation time:

- `TABLE` specifies that all distributions for the table be created at the table level.
- `FRAGMENT` specifies that distributions be created and maintained for each fragment.
- `AUTO` specifies that the database server apply criteria at run time to determine whether fragment-level distributions are necessary. These criteria require that all of the following conditions are true:

- The SYSSBSPACENAME configuration parameter setting specifies an existing sbspace.
- The table is fragmented by an EXPRESSION, INTERVAL, or LIST strategy.
- The table has more than a million rows.

If any of these criteria are not satisfied, the database server creates table-level distributions, rather than fragment-level.

These properties are always applied. If the STATLEVEL setting is AUTO, this setting overrides the default values.

Note: The SYSSBSPACENAME configuration parameter, which must be set when the database server instance is initialized, specifies the sbspace in which the database server stores fragment-level data distribution statistics. These statistics are stored as BLOB objects in the **encdist** column of the **syfragsdist** system catalog table. For the database server to support fragment level statistics, the SYSSBSPACENAME configuration parameter setting specifies an existing sbspace.

If you use the Statistics Options clause to set the STATLEVEL property to FRAGMENT, the database server returns an error if either of the following is true:

- The SYSSBSPACENAME configuration parameter is not set.
- The sbspace that SYSSBSPACENAME specifies was not properly allocated by the onspaces -c -S command.

CREATE TEMP TABLE statement

Use the CREATE TEMP TABLE statement to create a temporary table in the current database.

Syntax

(explicit id unique_443_Connect_42_ctt001a) unique_443_Connect_42_ctt001a(explicit id unique_443_Connect_42_ctt02a)
unique_443_Connect_42_ctt02a(explicit id unique_443_Connect_42_ctt003a) unique_443_Connect_42_ctt003a(explicit id unique_443_Connect_42_ctt04a)
unique_443_Connect_42_ctt04a(explicit id unique_443_Connect_42_ctt05a) unique_443_Connect_42_ctt05a(explicit id unique_443_Connect_42_ctt06a)
unique_443_Connect_42_ctt06a

```
CREATE { LARGE | SMALL } TEMP TABLE [IF NOT EXISTS] table ( <Column Definition> [ , {{
 <Multiple-Column Constraint Format> [ <Column Definition> ] }} ] ) [ WITH NO LOG ] [ <Storage Options> ] [ <LOCK MODE
Options> ] [ <USING Access-Method Clause> ]
```

Element	Description	Restrictions	Syntax
table	Name declared here for a table	Must be unique in session. See Declaring a name for a temporary table on page 371	Identifier on page 1231

Usage

You must have the Connect privilege on the database to create a temporary table. The temporary table is visible only to the user who created it.

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if a temporary table of the specified name already exists in the current session.

You can also define indexes and constraints on temporary tables that you define with the CREATE TEMP TABLE statement.

In , using the CREATE TEMP TABLE statement outside the CREATE SCHEMA statement generates warnings if you set **DBANSIWARN**.

In ESQL/C, the CREATE TEMP TABLE statement generates warnings if you use the **-ansi** flag or set the **DBANSIWARN** environment variable.

Declaring a name for a temporary table

A temporary table is associated with a session, not with a database. When you create a temporary table, you cannot create another temporary table with the same name (even for another database) until you drop the first temporary table or end the session.

The name of a temporary table must follow the naming requirements for SQL identifiers, but it cannot be a component of a qualified database-object name. When you create a temporary table with the CREATE TEMP TABLE statement, you cannot specify any authorization identifier as its *owner*. Unlike a permanent table, a temporary table cannot be referenced in an SQL statement by qualifying its identifier with an *owner* name, or a *database* name, or a *database server* name.

The name of a temporary table must be different from the name of any other table, view, sequence object, or synonym in the current database server. Otherwise, the temporary table takes precedence in your session over any permanent table of the same name. The name that you declare for the temporary table, however, need not be different from the temporary table names that are declared in other sessions of the same database server.

If you issue a cross-database DML statement that references a remote permanent table while your local database contains a temporary table with the same name, the DML statement accesses that local temporary table, rather than the remote permanent table.

Column definition

Use the Column Definition segment of the CREATE TEMP TABLE statement to declare the name and the data type (and optionally a default value and constraints) of a single column of the temporary table.

(explicit id unique_506_Connect_42_cold001b) unique_506_Connect_42_cold001b(explicit id unique_506_Connect_42_cold002b) unique_506_Connect_42_cold002b(explicit id unique_506_Connect_42_cold003b) unique_506_Connect_42_cold003b(explicit id unique_506_Connect_42_cold004b) unique_506_Connect_42_cold004b

Column Definition

" *column* "

" <Data Type>[] "

" { | [<DEFAULT Clause>[]] | [<Single-Column Constraint Format>[]] } "

Element	Description	Restrictions	Syntax
column	Name declared here for a column in the table	Must be unique in its table	Identifier on page 1231

This portion of the CREATE TEMP TABLE statement resembles the corresponding syntax segment in the CREATE TABLE statement. The differences include these:

- You cannot define a referential constraint on the column.
- The data type cannot be IDSSECURITYLABEL.
- The SECURED WITH *label* option is not supported for temporary tables.

Just as when you create permanent tables, any explicit or default storage size specification for a column of a built-in character type, such as CHAR, LVARCHAR, NCHAR, NVARCHAR, or VARCHAR, is interpreted in units of bytes, unless the SQL_LOGICAL_CHAR configuration parameter is set to enable logical character semantics for datatype declarations. See the *HCL® Informix® Administrator's Reference* for more information about the effect of the SQL_LOGICAL_CHAR setting in locales that support multibyte code sets, such as **UTF-8**, where a single logical character can require more than one byte of storage.

Single-Column Constraint Format

Use the single-column constraint format to create one or more data-integrity constraints for a single column in a temporary table.

(explicit id unique_510_Connect_42_sccfor001) unique_510_Connect_42_sccfor001(explicit id unique_510_Connect_42_sccfor002) unique_510_Connect_42_sccfor002

Single-Column Constraint Format

" { | { { **NULL** | **NOT NULL** } | **UNIQUE** | **DISTINCT** | **PRIMARY KEY** } | <CHECK Clause>[] } "

This is a subset of the syntax of Single-Column Constraint Format on page 327 that the CREATE TABLE statement supports.

You can find detailed discussions of specific constraints in these sections.

Constraint

For more information, see

CHECK

CHECK Clause on page 335

DISTINCT

Using UNIQUE or DISTINCT Constraints on page 329

NOT NULL

Using the NOT NULL Constraint on page 328

NULL

Using the NULL Constraint on page 329

PRIMARY KEY

Using the PRIMARY KEY Constraint on page 330

UNIQUE

Using UNIQUE or DISTINCT Constraints on page 329

Constraints that you define on temporary tables are always enabled.

Multiple-Column Constraint Format

Use the multiple-column constraint format to associate one or more columns with a constraint. This alternative to the single-column constraint format allows you to associate multiple columns with a constraint.

(explicit id unique_507_Connect_42_mccfor002) unique_507_Connect_42_mccfor002(explicit id unique_507_Connect_42_mccfor003)

unique_507_Connect_42_mccfor003

Multiple-Column Constraint Format

```
" { { { { NOT NULL | NULL } } | { UNIQUE | [ DISTINCT ] | PRIMARY KEY } ( column ) | <CHECK Clause> [ } "
```

Element	Description	Restrictions	Syntax
column	Name of column or columns on which the constraint is placed	Must be unique in a table, but the same name can be in different tables of the same database	Identifier on page 1231

This is a subset of the syntax of Multiple-Column Constraint Format on page 339 that the CREATE TABLE statement supports.

This alternative to the single-column constraint segment of CREATE TEMP TABLE can associate multiple columns with a constraint. Constraints that you define on temporary tables are always enabled.

The following table indicates where you can find detailed discussions of specific constraints.

Constraint	For more information, see	For an example, see
CHECK	CHECK Clause on page 335	Defining Check Constraints Across Columns on page 341
DISTINCT	Using UNIQUE or DISTINCT Constraints on page 329	Examples of the Multiple-Column Constraint Format on page 341
PRIMARY KEY	Using the PRIMARY KEY Constraint on page 330	Defining Composite Primary and Foreign Keys on page 341
UNIQUE	Using UNIQUE or DISTINCT Constraints on page 329	Examples of the Multiple-Column Constraint Format on page 341

See also the section Differences Between a Unique Constraint and a Unique Index on page 330.

Using the WITH NO LOG option

Use the WITH NO LOG option to reduce the overhead of transaction logging for the temporary table. If you specify WITH NO LOG, data manipulation language (DML) operations on the temporary table are not included in the transaction log records.

The WITH NO LOG keywords are required on all temporary tables that you create in temporary dbspaces. Within a cluster environment, the WITH NO LOG keywords are required when you create a temporary table on a secondary server.

If the ONCONFIG parameter TEMPTAB_NOLOG is set to `1`, logging of temporary tables is disabled and all temporary tables are non-logging by default. This setting can improve the performance of operations that use temporary tables, such as HDR operations. The WITH NO LOG option is not needed when the TEMPTAB_NOLOG setting has disabled logging of temporary tables. For information about how to set the TEMPTAB_NOLOG parameter, see your *HCL® Informix® Administrator's Reference*.

If you use the WITH NO LOG option in a database that does not use logging, the WITH NO LOG keywords of the CREATE TEMP TABLE statement have no effect. If your database does not support transaction logging, every table behaves as if the WITH NO LOG option were specified.

The ALTER TABLE statement cannot change the logging status of a temporary table. Once you turn off logging on a temporary table, you cannot turn it back on; a temporary table, therefore, is either always logged or else never logged.

The following temporary table is not logged in a database that uses transaction logging:

```
CREATE TEMP TABLE tab2 (fname CHAR(15), lname CHAR(15))
   WITH NO LOG;
```

Like all data definition language (DDL) statements of SQL, the `CREATE TEMP TABLE` statement above that creates tab2 is logged. The `WITH NO LOG` keywords, however, prevent transaction logging of any DELETE, INSERT, LOAD, MERGE, SELECT, UNLOAD, or UPDATE operations on tab2.

Storage options for temporary tables

Use the Storage Options segment of the CREATE TEMP TABLE statement to specify the storage location and distribution scheme for the table. This is an extension to the ANSI/ISO standard for SQL syntax.

(explicit id unique_508_Connect_42_stopt004) unique_508_Connect_42_stopt004(explicit id unique_508_Connect_42_stopt005)

unique_508_Connect_42_stopt005(explicit id unique_508_Connect_42_stopt006) unique_508_Connect_42_stopt006

Storage Options

" [{ IN { *dbspace* | *extspace* } | <FRAGMENT BY Clause> [] }] "

" [<PUT Clause> []] "

" [<EXTENT SIZE Options> []] "

Element	Description	Restrictions	Syntax
dbspace	Dbspace or temporary dbspace in which to store the temporary table.	Must already exist	Identifier on page 1231
extspace	Name that **onspaces** assigned to a storage area outside the database server	Must already exist	See documentation for access method.

Only temporary tables that include BLOB or CLOB columns can include the PUT clause as a storage option.

If you specify a temporary dbspace after the IN keyword, the database server does not perform any logical logging nor physical logging of the temporary table. You cannot mirror a temporary dbspace.

If you specify no extent size option, the default extent size is 8 pages.

To create a fragmented, unique index on a temporary table, you must specify an explicit expression-based distribution scheme for a temporary table in the CREATE TEMP TABLE statement. (Fragmentation of the index by ROUND ROBIN is not supported, and fragmentation by LIST or by INTERVAL is automatic, for a unique index on a table that uses a list or interval storage partitioning strategy.)

Where temporary tables are stored

The distribution scheme that you specify with the CREATE TEMP TABLE statement (either with the IN clause or the FRAGMENT BY clause) takes precedence over the information that the **DBSPACETEMP** environment variable or the DBSPACETEMP configuration parameter specifies.

If you do not specify an explicit distribution scheme for a temporary table, its storage location depends on the **DBSPACETEMP** environment variable (or DBSPACETEMP configuration parameter) setting.

- If **DBSPACETEMP** and DBSPACETEMP are not set, all temporary tables are created without fragmentation in the same dbspace where the database was created (or in **rootdbs**, if the database was not created in another dbspace).
- If only one dbspace for temporary tables is specified by **DBSPACETEMP** (or by DBSPACETEMP, if **DBSPACETEMP** is not set), all temporary tables are created without fragmentation in the specified dbspace.
- If **DBSPACETEMP** (or DBSPACETEMP, if **DBSPACETEMP** is not set) specifies two or more dbspaces for temporary tables, then each temporary table is created in one of the specified dbspaces.

  In a non-logging database, each temporary table is created in a temporary dbspace; in databases that support transaction logging, the temporary table is created in a standard dbspace. The database server tracks which of these dbspaces was most recently used, and when it receives the next request to allocate temporary storage, the database server uses the next available dbspace (in a round-robin pattern) to allocate I/O operations evenly among the dbspaces.

For example, if you create three temporary tables in a database with logging where **DBSPACETEMP** specifies **tempspc1**, **tempspc2**, and **tempspc3** as the default dbspaces for temporary tables, then the first table is created in the dbspace called **tempspc1**, the second table is created in **tempspc2**, and the third one is created in **tempspc3**, if these are the only requests for temporary storage.

Temporary tables created with SELECT INTO TEMP and WITH NO LOG are spread across the dbspaces listed in the DBSPACETEMP configuration parameter or **DBSPACETEMP** environment variable. Thus, the **DBSPACETEMP** (or DBSPACETEMP) settings that specify multiple dbspaces can result in round-robin fragmentation across all dbspaces in the temporary dbspace.

If you create a temporary table and specify WITH NO LOG, operations on the temporary table are not included in the transaction log records. If there is a logged space in the DBSPACETEMP list, the temporary table created with the SELECT .. INTO TEMP WITH NO LOG option is fragmented by a round-robin distribution scheme in the non-logged temporary dbspaces. For example, if from a list of 10 dbspaces, only one dbspace is logged, the table is fragmented by a round-robin distribution scheme in the 9 non-logged temporary dbspaces.

The following example shows how to insert data into a temporary table called **result_tmp** to output to a file the results of a user-defined function (**f_one**) that returns multiple rows:

```
CREATE TEMP TABLE result_tmp( ... );
INSERT INTO result_tmp EXECUTE FUNCTION f_one();
UNLOAD TO 'file' SELECT * FROM result_tmp;
```

Differences between temporary and permanent tables

Compared to permanent tables, temporary tables differ in these ways:

- They have fewer types of constraints available.
- They have fewer options that you can specify.
- They are not visible to other users or sessions.

- They do not appear in the system catalog tables.
- They are not preserved, as described in the section Duration of temporary tables on page 377.

The INFO statement and the **Info Menu** option of cannot reference temporary tables.

Duration of temporary tables

The duration of a temporary table depends on whether or not it is logged.

A logged temporary table exists until one of the following events occurs:

- The application disconnects.
- The DROP TABLE statement is issued on the temporary table.
- The database is closed.

When any of these events occurs, the temporary table is deleted.

Nonlogging temporary tables include tables that were created using the WITH NO LOG option of CREATE TEMP TABLE.

A nonlogging temporary table exists until one of the following situations occurs:

- The application disconnects.
- The DROP TABLE statement is issued on the temporary table.
- The database is closed, and the nonlogging temporary table includes at least one column of a user-defined type, or of a built-in opaque data type. (The Informix® built-in opaque data types include BLOB, BOOLEAN, BSON, CLOB, JSON, LVARCHAR, and IDSSECURITYLABEL.)

If the nonlogging temporary table does not include any columns of UDTs or of built-in opaque data types, you can use that table to transfer data from one database to another while the application remains connected, because the table is not destroyed when the database is closed. You must use a permanent table (or some other strategy) if the data to be transferred includes UDTs or built-in opaque data types.

TABLE_SIZE session environment option

The TABLE_SIZE session environment option controls the default size of newly created permanent tables for a session.

The SET ENVIRONMENT TABLE_SIZE statement of SQL supports the following syntax:

```
"SET ENVIRONMENT TABLE_SIZE {'SMALL' | 'LARGE'}"
```

Usage

Small tables are limited to 16.7 million pages of any supported size. Large tables make use of 8-byte row identifiers and support 140 trillion pages. Whether a table is created small or large affects its maximum size, but does not affect its initial size.

When a Version 15.0 instance is in native mode (Infrastructure 1 is enabled) newly created tables are large by default. That default can be changed for a session by setting TABLE_SIZE to 'SMALL'.

Explicit use of either the 'LARGE' or 'SMALL' keyword in the CREATE TABLE statement overrides the TABLE_SIZE session environment option.

The following message in an oncheck -pt output indicates that a table is large:

```
"TBLspace uses large rowids"
```

Related information

SMALL and LARGE tables on page 319

CREATE TRIGGER statement

Use the CREATE TRIGGER statement to define a trigger on a table. You can also use CREATE TRIGGER to define an INSTEAD OF trigger on a view.

This is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_192_Connect_42_crtr003) unique_192_Connect_42_crtr003(explicit id unique_192_Connect_42_crtr004)
unique_192_Connect_42_crtr004(explicit id unique_192_Connect_42_crtr005) unique_192_Connect_42_crtr005

```
CREATE TRIGGER [IF NOT EXISTS] [ <Owner Name> ] trigger { <Trigger on a Table> | INSTEAD OF <Trigger on a View> } { ENABLED | [DISABLED] }
```

Element	Description	Restrictions	Syntax
trigger	Name that you declare here for a new trigger	Must be unique among the names of triggers in the current database	Identifier on page 1231

Usage

A *trigger* is a database object that, unless disabled, automatically executes a specified set of SQL statements, called the *trigger action*, when a specified *trigger event* occurs.

The trigger event that initiates the trigger action can be an INSERT, DELETE, UPDATE, or a SELECT statement. The MERGE statement can also be the triggering event for an UPDATE, DELETE, or INSERT trigger. The event definition must specify the table or view on which the trigger is defined. (SELECT or UPDATE events for triggers on tables can also specify one or more columns.)

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if a trigger of the specified name is already defined on a table or view in the current database.

You can use the CREATE TRIGGER statement in two distinct ways:

- You can define a trigger on a table in the current database.
- You can also define an INSTEAD OF trigger on a view in the current database.

Any SQL statement that is an instance of the trigger event is called a *triggering statement*. When the event occurs, triggers defined on tables and triggers defined on views differ in whether the triggering statement is executed:

- For tables, the trigger event and the trigger action both execute.
- For views, only the trigger action executes, instead of the event.

That is, if you define an INSTEAD OF trigger on an updatable view, the database server does not perform the DML operation that corresponds to the *trigger event*, but instead performs the specified *trigger action* on the view or on its base table.

Important: In some cases, however, after the CREATE TRIGGER statement defines queries on a table or on a column as triggering events for enabled Select triggers, the database server does not execute the trigger action when the query executes. For a list of contexts where this is the expected behavior, see Circumstances When a Select Trigger Is Not Activated on page 392. For a similar list of contexts where the trigger actions of enabled Insert, Delete, or Update triggers on views are not executed, see Restrictions on INSTEAD OF Triggers on Views.

The CREATE TRIGGER statement can support the integrity of data in the database by defining rules by which specified DML operations (the triggering events) cause the database server to take specified actions. The following sections describe the syntax elements.

Clause	Page	Effect
Defining a Trigger Event and Actions	Defining a Trigger Event and Action on page 380	Associates triggered actions with an event
Trigger Modes	Trigger Modes on page 383	Enables or disables the trigger
Insert Events and Delete Events	INSERT Events and DELETE Events on page 386	Defines Insert events and Delete events
Update Events	UPDATE Event on page 387	Defines Update events
Select Events	SELECT Event on page 388	Defines Select events
Action Clause	Action Clause on page 392	Defines triggered actions
REFERENCING Clause for Delete	REFERENCING Clause for Delete on page 395	Declares qualifier for deleted values
REFERENCING Clause for Insert	REFERENCING Clause for Insert triggers on page 396	Declares qualifier for inserted values
REFERENCING Clause for Update	REFERENCING Clause for Update on page 397	Declares qualifiers for old and new values

Clause	Page	Effect
REFERENCING Clause for Select	REFERENCING Clause for Select on page 398	Declares qualifier for result set values
Correlated Table Action	Correlated Table Action on page 399	Defines triggered actions
Triggered Action	Triggered Action on a Table on page 399	Defines triggered actions
INSTEAD OF Trigger on Views	#unique_516	Defines a trigger on views
Action Clause of INSTEAD OF Triggers	The Action Clause of INSTEAD OF Triggers	Triggered actions on views

Defining a Trigger Event and Action

This syntax defines the event and action of a trigger on a table or on a view.

(explicit id unique_515_Connect_42_tea003) unique_515_Connect_42_tea003(explicit id unique_515_Connect_42_tea004)
unique_515_Connect_42_tea004(explicit id unique_515_Connect_42_tea005) unique_515_Connect_42_tea005(explicit id unique_515_Connect_42_tea006)
unique_515_Connect_42_tea006(explicit id unique_515_Connect_42_tea007) unique_515_Connect_42_tea007

Trigger on a Table

" { { **DELETE** | **SELECT** [**OF** *column*] } **ON***table* <DELETE and SELECT Clauses> | **UPDATE** [**OF** *column*] **ON***table* <UPDATE Clause> | **INSERTON***table* { <NEW Declaration>[] <Correlated Action>[] | <Action Clause>[] } } "

DELETE and SELECT Clauses

" { <Action Clause>[] | <OLD Declaration>[] <Correlated Action>[] } "

UPDATE Clause

" { <Action Clause>[] | <OLD Declaration>[] [<NEW Declaration>[]] <Correlated Action>[] } "

Trigger on a View

" { **INSERT ON***view* [**REFERENCING NEW** [**AS**] *correlation*] | **DELETE ON***view* [**REFERENCINGOLD** [**AS**] *correlation*] | **UPDATE ON***view* [{ [**REFERENCING OLD [AS]** *correlation* [**NEW [AS]** *correlation*]] | **REFERENCINGNEW [AS]** *correlation* [**OLD [AS]** *correlation*] }] } "

" **FOR EACH ROW** "

" <INSTEAD OF Triggered Action>[] "

Element	Description	Restrictions	Syntax
column	The name of a column in the triggering *table*	Must exist	Identifier on page 1231
correlation	Name that you declare here to qualify an old or new column value (as *correlation*.*column*) in a triggered action	Must be unique in this trigger	Identifier on page 1231
table, view	Name or synonym of the triggering table or view. The *table* or *view* can include an *owner*. qualifier.	Must exist in the current database	Identifier on page 1231

The left-hand portion of the main diagram (including the *table* or *view*) defines the *trigger event* (sometimes called the *triggering event*). The rest of the diagram declares correlation names and defines the *trigger action* (sometimes called the *triggered action*).

For the syntax to specify the triggered action of triggers on tables, see Action Clause on page 392 and Correlated Table Action on page 399.

For the syntax to specify the triggered action of INSTEAD OF triggers on views, see The Action Clause of INSTEAD OF Triggers.

Restrictions on Triggers

To create a trigger on a table (or an INSTEAD OF trigger on a view), you must own the table or view, or have DBA privilege. For the relationship between the privileges of the trigger owner and those of other users, see Privileges to Execute Trigger Actions on page 408.

The table on which you create a trigger must exist in the current database. You cannot create a trigger on any of the following types of tables:

- A diagnostics table, a violations table, or a table in another database
- A temporary table or a system catalog table
- A table object that the CREATE EXTERNAL TABLE or CREATE SEQUENCE statement created.

In , if you want to define a trigger as part of a schema, place the CREATE TRIGGER statement inside a CREATE SCHEMA statement.

If you are embedding the CREATE TRIGGER statement in programs, you cannot use a host variable in the trigger definition.

You can use the DROP TRIGGER statement to remove an existing trigger. If you use the DROP TABLE or DROP VIEW statement to remove triggering tables or views from the database, all triggers on those tables or views are also dropped.

The ON EXCEPTION statement of SPL has no effect when it is issued from a trigger routine, nor from the Action clause or the Correlated Action clause of a trigger.

The triggered action of an Insert trigger that increments a BIGSERIAL, SERIAL, or SERIAL8 column does not update the **sqlca.sqlerrd[1]** field of the SQL Communication Area structure. The triggered INSERT operation can successfully increment the serial counter for the column, but the value of the **sqlca.sqlerrd[1]** field remains zero, rather than being reset to the new serial value.

You cannot define a DELETE trigger on a table that has a referential constraint that specifies ON DELETE CASCADE.

UNION subqueries cannot be triggering events. If a valid UNION subquery specifies a column on which a Select trigger has been defined, the query succeeds, but the trigger (or the INSTEAD OF trigger on a view) is ignored.

The database server cannot use parallel processing for some triggered actions. PDQ is automatically disabled in the FOR EACH ROW section for any DML statement that corresponds to the type of triggering event:

- SELECT statements in the Action clause of a Select trigger
- DELETE or MERGE statements in the Action clause of a Delete trigger
- INSERT or MERGE statements in the Action clause of an Insert trigger
- UPDATE or MERGE statements in the Action clause of an Update trigger.

The scope of this restriction on PDQ is the FOR EACH ROW section. It has no effect on DML statements in the BEFORE or AFTER sections of the Action clause.

For additional restrictions on INSTEAD OF triggers on views, see Restrictions on INSTEAD OF Triggers on Views.

Trigger Modes

You can set a trigger mode to enable or disable a trigger when you create it.

Trigger Modes

```
{ DISABLED | ENABLED }
```

You can create triggers on tables or on views in ENABLED or DISABLED mode.

- When a trigger is created in ENABLED mode, the database server executes the trigger action when the trigger event is encountered. (If you specify no mode when you create a trigger, ENABLED is the default mode.)
- When a trigger is created in DISABLED mode, the trigger event does not cause execution of the trigger action. In effect, the database server ignores the trigger and its action, even though the **systriggers** system catalog table maintains information about the disabled trigger.

You can also use the SET TRIGGERS option of the Database Object Mode statement to set an existing trigger to the ENABLED or DISABLED mode.

After a DISABLED trigger is enabled by the SET TRIGGERS statement, the database server can execute the trigger action when the trigger event is encountered, but the trigger does not perform retroactively. The database server does not attempt to execute the trigger for rows that were selected, inserted, deleted, or updated while the trigger was disabled and before it was enabled.

Warning: Because the behavior of a trigger varies according to its ENABLED or DISABLED mode, be cautious about disabling a trigger. If disabling a trigger will eventually destroy the semantic integrity of the database, do not disable the trigger.

Trigger Inheritance in a Table Hierarchy

By default, any trigger that you define on a typed table of Informix® is inherited by all its subtables.

In versions of Informix® earlier than version 11.10, however, if you define a trigger on a subtable of a typed table, that trigger overrides any trigger for the same type of triggering event (Select, Delete, Insert, or Update) that the subtable inherits from

its supertable. In this version of Informix®, however, a table can inherit more than one trigger that the same triggering event activates, so both triggers are defined for the same type of event on the subtable.

In all versions of Informix®, a trigger that you set on a subtable is inherited by all its dependent tables, but has no effect on its supertable.

This behavior is important when you require a trigger to be enabled in a supertable, but to be disabled in its subtable. In Informix® 10.00 and in earlier versions, you cannot use the SET TRIGGERS option of the SET Database Object Mode statement to disable an inherited trigger selectively within a hierarchy. In this release, however, disabling a trigger on a table within a table hierarchy has no effect on inherited triggers. For example, the following statement has no effect on triggers on table objects that are above or below *subtable* within a table hierarchy:

```
SET TRIGGERS FOR subtable DISABLED
```

Similarly, the DROP TRIGGER statement cannot destroy an inherited trigger without also destroying the trigger on the supertable. In this situation, you must instead define a trigger with no Action clause on the subtable. Because triggers are not additive, this empty trigger overrides the inherited trigger and executes for the subtable and for any subtables under the subtable, which are not subject to further overrides.

Triggers and SPL Routines

You cannot define a trigger in an SPL routine that is called inside a DML (data manipulation language) statement, as listed in Data Manipulation Language Statements on page 11. Thus, the following statement returns an error if the **sp_items** procedure includes the CREATE TRIGGER statement:

```
INSERT INTO items EXECUTE PROCEDURE sp_items;
```

You can use the CREATE FUNCTION or CREATE PROCEDURE statement of SQL with the REFERENCING clause to define *trigger routines* that include the FOR *table* or FOR *view* specification. These UDRs must include the REFERENCING clause that declares *correlation* names for OLD or NEW column values in the specified *table* or *view*. Triggers on the *table* or *view* can invoke the trigger routine from the FOR EACH ROW section of the Triggered Action list. Triggers can also invoke non-trigger routines from the BEFORE and AFTER sections of the Triggered Action list, but these UDRs cannot use *correlation* names to reference the NEW or OLD column values. The REFERENCING clause in a trigger routine supports the same syntax as in the CREATE TRIGGER statement, as described in the section REFERENCING Clauses on page 395.

Multiple triggers that the same triggering event executes can invoke more than one trigger routine, and these trigger routines can access the same NEW or OLD column values by using SPL variables that have the same names or different names. When a single triggering event executes multiple triggers, the order of execution is not guaranteed, but all of the BEFORE triggered actions execute before any of the FOR EACH ROW triggered actions, and all of the AFTER triggered actions execute after all of the FOR EACH ROW triggered actions.

For UDRs that are not trigger routines, SPL variables are not valid in CREATE TRIGGER statements. An SPL routine cannot perform INSERT, DELETE, or UPDATE operations on any table or view that is not local to the current database. See also Rules for SPL Routines on page 407 for additional restrictions on SPL routines that are invoked in triggered actions.

Trigger Events

The *trigger event* specifies what DML statements can initiate the trigger. The event can be an INSERT, DELETE, or UPDATE operation on the *table* or *view*, or a SELECT operation that queries the *table*. Each CREATE TRIGGER statement must specify exactly one trigger event. Any SQL statement that is an instance of the trigger event is called a *triggering statement.*

For each *table*, you can define triggers that are activated by INSERT, DELETE, UPDATE, or SELECT statements. For each *view*, you can define INSTEAD OF triggers that are activated by INSERT, DELETE, or UPDATE statements. Multiple triggers on the same table or view can be activated by different types of trigger events or by the same type of trigger event.

You cannot specify a DELETE event if the triggering table has a referential constraint that specifies ON DELETE CASCADE.

You are responsible for guaranteeing that the triggering statement returns the same result with and without the trigger action on a table. See also the sections Action Clause on page 392 and Triggered Action on a Table on page 399.

A triggering statement from an external database server can activate the trigger.

As the following example shows, an Insert trigger on **newtab**, managed by **dbserver1**, is activated by an INSERT statement from **dbserver2**. The trigger executes as if the INSERT originated on **dbserver1**.

```
-- Trigger on stores_demo@dbserver1:newtab
CREATE TRIGGER ins_tr INSERT ON newtab
   REFERENCING new AS post_ins
   FOR EACH ROW(EXECUTE PROCEDURE nt_pct (post_ins.mc));
-- Triggering statement from dbserver2
INSERT INTO stores_demo@dbserver1:newtab
   SELECT item_num, order_num, quantity, stock_num, manu_code,
   total_price FROM items;
```

Informix® also supports INSTEAD OF triggers on views, which are initiated when a triggering DML operation references the specified view. The INSTEAD OF trigger replaces the trigger event with the specified trigger action on a view, rather than execute the triggering INSERT, DELETE, or UPDATE operation. A *view* can have any number of INSTEAD OF trigger defined for each type of INSERT, DELETE, or UPDATE triggering event.

Trigger Events with Cursors

For triggers on tables, if the triggering statement uses a cursor, each part of the trigger action (including BEFORE, FOR EACH ROW, and AFTER, if these are specified for the trigger) is activated for each row that the cursor processes.

This behavior differs from what occurs when a triggering statement does not use a cursor and updates multiple rows. In this case, any BEFORE and AFTER triggered actions execute only once, but the FOR EACH ROW action list is executed for each row processed by the triggering statement. For additional information about trigger actions, see Action Clause on page 392

Privileges on the Trigger Event

You must have appropriate Insert, Delete, Update, or Select privilege on the triggering table or view to execute a triggering INSERT, DELETE, UPDATE, or SELECT statement as the trigger event. The triggering statement might still fail, however, if you do not also have the privileges necessary to execute one of the SQL statements in the trigger action. When the trigger

actions are executed, the database server checks your privileges for each SQL statement in the trigger definition, as if the statement were being executed independently of the trigger. For information on the privileges needed to execute the trigger actions, see Privileges to Execute Trigger Actions on page 408.

Performance Impact of Triggers

The INSERT, DELETE, UPDATE, and SELECT statements that initiate triggers might appear to execute slowly because they execute additional SQL statements, and the user might not know that other actions are occurring.

The execution time for a trigger event depends on the complexity of the trigger action and whether it initiates other triggers. The time increases as the number of cascading triggers increases. For more information on triggers that initiate other triggers, see Cascading Triggers on page 410.

INSERT Events and DELETE Events

INSERT and DELETE events on tables are defined by those keywords and by the ON *table* clause, using the following syntax.

INSERT or DELETE Event on a Table

" { **INSERT** | **DELETE** } "

" **ON***table* "

Element	Description	Restrictions	Syntax
table	Name of the triggering table	Must exist in the database	Identifier on page 1231

An Insert trigger is activated when an INSERT statement includes the specified *table* (or a synonym for *table*) in its INTO clause. Similarly, a Delete trigger is activated when a DELETE statement includes the specified *table* (or a synonym for *table*) in its FROM clause.

The MERGE statement can also activate an Insert trigger, if the specified *table* of the Insert trigger is the target table of a MERGE statement that includes the Insert clause. Similarly, the MERGE statement can also activate a Delete trigger, if the specified *table* of the Delete trigger is the target table of a MERGE statement that includes the Delete clause.

The TRUNCATE TABLE statement does not activate Delete triggers when it removes all the rows from a table. If an enabled Delete trigger is defined for a table on which you do not hold the Alter privilege, the database server returns an error if you attempt to truncate that table, even though the TRUNCATE statement cannot be the triggering event for a Delete trigger. (For more information about the discretionary access privileges that truncate operations require, see the TRUNCATE statement on page 858.)

For triggers on views, the INSTEAD OF keywords must immediately precede the INSERT, DELETE, or UPDATE keyword that specifies the type of trigger event, and the name or synonym of a *view* (rather than of a table) must follow the ON keyword. The section #unique_516 describes the syntax for defining INSTEAD OF trigger events.

Any number of Insert triggers, and any number of Delete triggers, can be defined on the same table.

If you define a trigger on a subtable within a table hierarchy, and the subtable supports cascading deletes, then a DELETE operation on the supertable activates the Delete triggers on the subtable.

See also the section Re-Entrancy of Triggers on page 404 for information about dependencies and restrictions on the actions of Insert triggers and Delete triggers.

UPDATE Event

UPDATE events (and SELECT events) can include an optional *column* list.

UPDATE Event

" UPDATE "

" [OF *column*] "

" ON*table* "

Element	Description	Restrictions	Syntax
column	Column that activates the trigger	Must exist in the triggering table	Identifier on page 1231
table	Name of the triggering table	Must exist in the database	Identifier on page 1231

The *column* list is optional. If you omit the OF *column* list, updating any column of *table* activates the trigger.

The OF *column* clause is not valid for an INSTEAD OF trigger on a view.

An UPDATE on the triggering table can activate the trigger in two cases:

- The UPDATE statement references any column in the *column* list.
- The UPDATE event definition has no OF *column* list specification.

Whether it updates one column or more than one column from the *column* list, a triggering UPDATE statement activates each Update trigger only once.

The MERGE statement can also activate an Update trigger, if the specified *table* of a trigger with no *columns* list is the target table of the MERGE statement, or if the Update clause of the MERGE statement references a column in the *column* list of the Update trigger.

Defining Multiple Update Triggers

Multiple Update triggers on a table can include the same or different columns. In the following example, **trig3** on the **items** table includes in its column list **stock_num**, which is a triggering column in **trig1**.

```
CREATE TRIGGER trig1 UPDATE OF item_num, stock_num ON items
   REFERENCING OLD AS pre NEW AS post
   FOR EACH ROW(EXECUTE PROCEDURE proc1());
CREATE TRIGGER trig2 UPDATE OF manu_code ON items
   BEFORE(EXECUTE PROCEDURE proc2());
CREATE TRIGGER trig3 UPDATE OF order_num, stock_num ON items
   BEFORE(EXECUTE PROCEDURE proc3());
```

When an UPDATE statement updates multiple columns that have different triggers, the firing order is based on the lowest-numbered column that is defined in the triggers that actually fired, regardless of whether that lowest-numbered column was actually the triggering column when the trigger fired. If several Update triggers are set on the same column or on the same set of columns, however, the order of trigger execution is not guaranteed.

The following example shows that table **taba** has four columns (**a**, **b**, **c**, **d**):

```
CREATE TABLE taba (a int, b int, c int, d int);
```

Define **trig1** as an update on columns **a** and **c**, and define **trig2** as an update on columns **b** and **d**, as the following example shows:

```
CREATE TRIGGER trig1 UPDATE OF a, c ON taba
   AFTER (UPDATE tabb SET y = y + 1);

CREATE TRIGGER trig2 UPDATE OF b, d ON taba
   AFTER (UPDATE tabb SET z = z + 1);
```

The following example shows a triggering statement for the Update trigger:

```
UPDATE taba SET (b, c) = (b + 1, c + 1);
```

Then **trig1** for columns **a** and **c** executes first, and **trig2** for columns **b** and **d** executes next. In this case, the smallest column number in the two triggers is column 1 (**a**), and the next is column 2 (**b**).

SELECT Event

DELETE and INSERT events are defined by those keywords (and the ON *table* clause), but SELECT and UPDATE events also support an optional *column* list.

SELECT Event

" **SELECT** "

" [**OF** *column*] "

" **ON** [*'owner'.*] *table* "

Element	Description	Restrictions	Syntax
column	Column that activates the trigger	Must exist in the triggering *table*	Identifier on page 1231

Element	Description	Restrictions	Syntax
owner	Owner of *table*	Must own *table*	Owner name on page 1262
table	Name of the triggering table	Must exist in the database	Identifier on page 1231

If you define more than one Select trigger on the same table, the *column* list is optional, and the *column* lists for each trigger can be unique or can duplicate that of another Select trigger.

A SELECT on the triggering table can activate the trigger in two cases:

- The SELECT statement references any column in the *column* list.
- The SELECT event definition has no OF *column* list specification.

(Sections that follow, however, describe additional circumstances that can affect whether or not a SELECT statement activates a Select trigger.)

Whether it specifies one column or more than one column from the *column* list, a triggering SELECT statement activates the Select trigger only once.

The action of a Select trigger cannot include an UPDATE, INSERT, or DELETE on the triggering table. The action of a Select trigger can include UPDATE, INSERT, and DELETE actions on tables other than the triggering table. The following example defines a Select trigger on one column of a table:

```
CREATE TRIGGER mytrig
   SELECT OF cola ON mytab REFERENCING OLD AS pre
   FOR EACH ROW (INSERT INTO newtab VALUES('for each action'));
```

You cannot specify a SELECT event for an INSTEAD OF trigger on a view.

Circumstances When a Select Trigger Is Activated

A query on the triggering table activates a Select trigger in these cases:

- The SELECT statement is a stand-alone SELECT statement.
- The SELECT statement occurs within a UDR called in a select list.
- The SELECT statement is a subquery in the Projection list.
- The SELECT statement is a subquery in the FROM clause.
- The SELECT statement occurs within a UDR called by EXECUTE PROCEDURE or EXECUTE FUNCTION.
- The SELECT statement selects data from a supertable in a table hierarchy. In this case the SELECT statement activates Select triggers for the supertable and all the subtables in the hierarchy.

For information on SELECT statements that do not activate a Select trigger, see Circumstances When a Select Trigger Is Not Activated on page 392.

Stand-alone SELECT Statements

A Select trigger is activated if the triggering column appears in the select list of the Projection clause of a stand-alone SELECT statement.

For example, if a Select trigger is defined to execute whenever column **col1** of table **tab1** is selected, then both of the following stand-alone SELECT statements activate the Select trigger:

```
SELECT * FROM tab1;
SELECT col1 FROM tab1;
```

SELECT Statements Within UDRs in the Select List

A Select trigger is activated by a UDR if the UDR contains a SELECT statement within its statement block, and the UDR also appears in the Projection clause of a SELECT statement.

For example, assume that an enabled Select trigger is defined on the table **tab1**, and that a UDR named **my_rtn** contains this SELECT statement in its statement block:

```
SELECT col1 FROM tab1;
```

Now suppose that the following SELECT statement invokes the **my_rtn** UDR in its projection list:

```
SELECT my_rtn() FROM tab2;
```

This SELECT statement activates the Select trigger defined on column **col1** of table **tab1** when the **my_rtn** UDR is executed.

UDRs that EXECUTE PROCEDURE or EXECUTE FUNCTION call

A Select trigger is activated by a UDR if the UDR contains a SELECT statement within its statement block that queries the table, and the UDR is called by an EXECUTE PROCEDURE or the EXECUTE FUNCTION statement.

For example, assume that an enabled Select trigger is defined on the table **tab1**, and that the user-defined routine named **my_rtn** contains the following SELECT statement in its statement block:

```
SELECT col1 FROM tab1;
```

Now suppose that the following statement invokes the **my_rtn** routine:

```
EXECUTE PROCEDURE my_rtn();
```

This statement activates the Select trigger defined on column **col1** of table **tab1** when the SELECT statement within the statement block is executed.

Subqueries in the Select List

A Select trigger can be activated by a subquery that appears in the select list of the Projection clause of a SELECT statement.

For example, if an enabled Select trigger is defined on **col1** of **tab1**, the subquery in the following SELECT statement activates that trigger:

```
SELECT (SELECT col1 FROM tab1 WHERE col1=1), colx, coly FROM tabz;
```

Subqueries in the FROM Clause of SELECT

Table expressions in the FROM clause of a SELECT statement can be the triggering event on a table that is referenced by an uncorrelated subquery. In the following example, the subquery that specifies a table expression is the triggering event for any enabled Select triggers that are defined on **col1** of **tab1**:

```
SELECT vcol FROM (SELECT FIRST 5 col1 FROM tab1 ORDER BY col1 ) vtab(vcol);
```

Subqueries in the WHERE Clause of DELETE or UPDATE

Subqueries that are specified using the Condition with Subquery syntax in the WHERE clause of the DELETE statement or the UPDATE statement cannot be the triggering event for a Select trigger.

In the following example, the subquery is not the triggering event for any enabled Select triggers that are defined on **col2** of **tab1**:

```
DELETE tab1 WHERE EXISTS
   (SELECT col2 FROM tab1 WHERE col2 > 1024);
```

The DELETE operation in same example, however, activates any enabled Delete triggers that are defined on tab1. No enabled Select trigger on tab1 can be activated by a subquery within a DELETE statement that modifies a table referenced in the FROM clause of the subquery.

Similarly, the subquery in the WHERE clause of the following UPDATE statement is not the triggering event for any enabled Select triggers that are defined on **col3** of **tab1**:

```
UPDATE tab1 SET col3 = col3 + 10
   WHERE col3 > ANY
      (SELECT col3 from tab1 WHERE col3 > 1);
```

The same example activates any enabled Update trigger that is defined on col3 of tab1, but no Select trigger can be updated by the subquery. For additional restrictions on Select triggers, see Circumstances When a Select Trigger Is Not Activated on page 392.

Select Triggers in Table Hierarchies

A subtable in the Informix® database inherits the Select triggers that are defined on its supertable. When you select from a supertable, the SELECT statement activates the Select triggers on the supertable and the inherited Select triggers on the subtables in the table hierarchy.

For example, assume that table **tab1** is the supertable and table **tab2** is the subtable in a table hierarchy. If the Select trigger **trig1** is defined on table **tab1**, a SELECT statement on table **tab1** activates the Select trigger **trig1** for the rows in table **tab1** and the inherited Select trigger **trig1** for the rows in table **tab2**.

If you add a Select trigger to a subtable, this Select trigger does not override the Select trigger that the subtable inherits from its supertable, but increases the number of Select triggers on the subtable. For example, if the Select trigger **trig1** is defined on column **col1** in supertable **tab1**, the subtable **tab2** inherits this trigger. If you define a Select trigger named **trig2** on column **col1** in subtable **tab2**, and a SELECT statement selects from **col1** in supertable **tab1**, this SELECT statement activates trigger **trig1** for the rows in table **tab1** and both triggers **trig1** and **trig2** for the rows in table **tab2**.

Circumstances When a Select Trigger Is Not Activated

A SELECT statement on the triggering table does not activate a Select trigger in certain circumstances:

- If a subquery or UDR that contains the triggering SELECT statement appears in any clause of a SELECT statement other than the Projection clause or the FROM clause, the Select trigger is not activated.

  For example, if the subquery or UDR appears in the WHERE clause or HAVING clause of a SELECT statement, the SELECT statement within the subquery or UDR does not activate the Select trigger.
- If the trigger action of a Select trigger calls a UDR that includes a triggering SELECT statement, the Select trigger on the SELECT in the UDR is not activated. Cascading Select triggers are not supported.
- If a SELECT statement contains a built-in aggregate or user-defined aggregate in its Projection clause, the Select trigger is not activated. For example, the following SELECT statement does not activate a Select trigger defined on **col1** of **tab1**:

  ```
  SELECT MIN(col1) FROM tab1;
  ```
- A SELECT statement that includes a set operator (including INTERSECT, MINUS, EXCEPT, UNION, or UNION ALL) does not activate a Select trigger.
- The SELECT clause of INSERT does not activate a Select trigger.
- A subquery in the WHERE clause of the DELETE or UPDATE statement cannot activate a Select trigger on the same table that the DELETE or UPDATE statement is modifying.
- If the Projection clause of a SELECT includes the DISTINCT or UNIQUE keywords, the SELECT statement does not activate a Select trigger.
- Select triggers are not supported on scroll cursors.
- If a SELECT statement refers to a remote triggering table, the Select trigger is not activated on the remote database server.
- Columns in the ORDER BY list of a query activate no Select triggers (nor any other triggers) unless they are also listed in the Projection clause.

An exception to the last restriction is that a Select trigger can be activated by a column in the ORDER BY list of a subquery in the FROM clause, whether or not the same column also appears in the Projection clause. In the following example, a table expression that includes **col1** in the ORDER BY clause (but not in the select list of the Projection clause) is the triggering event for any enabled Select triggers that are defined on **col1** of **tab1**:

```
SELECT vcol FROM (SELECT col2 FROM tab1 ORDER BY col1 ) vtab(vcol);
```

Action Clause

The Action clause defines the SQL statements that are executed when the trigger is activated. For a trigger on a table, there can be up to three sections in the Action clause: BEFORE, AFTER and FOR EACH ROW.

- The BEFORE actions are executed once for each triggering event, before the database server performs the triggering DML operation.
- The AFTER actions are also executed once for each triggering DML event, after the operation on the table is complete, in the context of the triggering statement.
- The FOR EACH ROW actions are executed for each row that is inserted, updated, deleted or selected in the DML operation, after the DML operation is executed on each row, but before the database server writes the values into the log and into the table.

If the same table has multiple triggers that are activated by the same triggering event, the order of trigger execution is not guaranteed, but all of the BEFORE triggered actions execute before any of the FOR EACH ROW triggered actions, and all of the AFTER triggered actions execute after all of the FOR EACH ROW triggered actions.

When you define an INSTEAD OF trigger on a view, the BEFORE and AFTER keywords are not supported, but the FOR EACH ROW section of the Action clause is valid. See the section #unique_516 for the syntax of specifying triggered actions on a view.

The Action clause has the following syntax.

(explicit id unique_523_Connect_42_acl001) unique_523_Connect_42_acl001

Action Clause

" [**BEFORE** <Triggered Action>[]] "

" [**FOR EACH ROW** <Triggered Action>[]] "

" [**AFTER** <Triggered Action>[]] "

For the trigger to have any effect on the table, you must define at least one triggered action, using the keywords BEFORE, FOR EACH ROW, or AFTER to indicate when the action occurs relative to execution of the triggering event.

You can specify actions for any or all of these three options on a single trigger, but any BEFORE action list must be specified first, and any AFTER action list must be specified last. For more information on the Action clause when a REFERENCING clause is also specified, see Correlated Table Action on page 399.

BEFORE Actions

The list of BEFORE trigger actions execute once before the triggering statement executes. Even if the triggering statement does not process any rows, the database server executes the BEFORE trigger actions.

FOR EACH ROW Actions

After a row of the triggering table is processed, the database server executes all of the statements of the FOR EACH ROW trigger action list; this cycle is repeated for every row that the triggering statement processes. (But if the triggering statement does not insert, delete, update, or select any rows, the database server does not execute the FOR EACH ROW trigger actions.)

The FOR EACH ROW action list of a Select trigger is executed once for each instance of a row. For example, the same row can appear more than once in the result of a query joining two tables. For more information on FOR EACH ROW actions that reference specific values in rows that the triggering statement processes, see REFERENCING Clauses on page 395.

As noted in the section Restrictions on Triggers on page 382, parallel data processing is disabled in FOR EACH ROW trigger actions for DML statements that correspond to the type of trigger event. For example, the database server does not apply PDQ to UPDATE statements in the FOR EACH ROW section of the Action clause of an Update trigger, nor to DELETE statements in the FOR EACH ROW section of the Action clause of a Delete trigger. This restriction on PDQ processing does not apply to DML statements in the BEFORE or AFTER sections of the Action clause.

AFTER Actions

The specified set of AFTER trigger actions executes once after the action of the triggering statement is complete. If the triggering statement does not process any rows, the AFTER trigger actions still execute.

Actions of Multiple Triggers

When an UPDATE or MERGE statement activates multiple triggers, the trigger actions merge. Assume that **taba** has columns **a**, **b**, **c**, and **d**, as this example shows:

```
CREATE TABLE taba (a INT, b INT, c INT, d INT);
```

Next, assume that you define **trig1** on columns **a** and **c**, and **trig2** on columns **b** and **d**. If both triggers specify BEFORE, FOR EACH ROW, and AFTER actions, then the trigger actions are executed in the following order:

1. BEFORE action list for trigger (**a**, **c**)
2. BEFORE action list for trigger (**b**, **d**)
3. FOR EACH ROW action list for trigger (**a**, **c**)
4. FOR EACH ROW action list for trigger (**b**, **d**)
5. AFTER action list for trigger (**a**, **c**)
6. AFTER action list for trigger (**b**, **d**)

The database server treats all the triggers that are activated by the same triggering statement as a single trigger, and the trigger action is the merged-action list. All the rules that govern a trigger action apply to the merged list as one list, and no distinction is made between the two original triggers.

Guaranteeing Row-Order Independence

In a FOR EACH ROW triggered-action list, the result might depend on the order of the rows being processed. You can ensure that the result is independent of row order by following these suggestions:

- Avoid selecting the triggering table in the FOR EACH ROW section.

  If the triggering statement affects multiple rows in the triggering table, the result of the SELECT statement in the FOR EACH ROW section varies as each row is processed. This condition also applies to any cascading triggers. See Cascading Triggers on page 410.
- In the FOR EACH ROW section, avoid updating a table with values derived from the current row of the triggering table.

  If the trigger actions modify any row in the table more than once, the final result for that row depends on the order in which rows from the triggering table are processed.
- Avoid modifying a table in the FOR EACH ROW section that is selected by another statement in the same FOR EACH ROW trigger action, including any cascading trigger actions.

If FOR EACH ROW actions modify a table, the changes might not be complete when a subsequent action of the trigger refers to the table. In this case, the result might differ, depending on the order in which rows are processed.

The database server does not enforce rules to prevent these situations because doing so would restrict the set of tables from which a trigger action can select. Furthermore, the result of most trigger actions is independent of row order. Consequently, you are responsible for ensuring that the results of the trigger actions are independent of row order.

REFERENCING Clauses

The REFERENCING clause for any event declares a *correlation* name (or for Update triggers, two *correlation* names) that can be used to qualify column values in the triggering table. These names enable FOR EACH ROW actions to reference new or old column values in the result of trigger events.

They also enable FOR EACH ROW actions to reference old column values that existed in the triggering table prior to modification by trigger events.

Correlation names are not valid if the triggered action includes both the INSERT statement and the BEFORE WHEN or AFTER WHEN keywords. This restriction does not affect triggered actions that specify the FOR EACH ROW keywords without the BEFORE or AFTER keywords, or that include no INSERT statement.

The REFERENCING clause syntax that is described here for the CREATE TRIGGER statement is also valid in CREATE FUNCTION and CREATE PROCEDURE statements that define a trigger routine, provided that the CREATE FUNCTION or CREATE PROCEDURE statement also includes the FOR *table_object* clause to specify the table or view whose FOR EACH ROW actions can invoke the trigger routine.

REFERENCING Clause for Delete

The REFERENCING clause for a Delete trigger can declare a correlation name for the deleted value in a column.

REFERENCING Clause for Delete

" REFERENCINGOLD "

" [AS] "

" *correlation* "

Element	Description	Restrictions	Syntax
correlation	Name that you declare here to qualify an old column value (as *correlation*.*column*) in a triggered action	Must be unique within this CREATE TRIGGER statement	Identifier on page 1231

The *correlation* is a qualifier for the column value in the triggering table before the triggering statement executed. The *correlation* is in scope in the FOR EACH ROW trigger action list. See Correlated Table Action on page 399.

To use a correlation name in a trigger action to refer to an old column value, prefix the column name with the correlation name and a period (**.**) symbol. For example, if the NEW *correlation* name is **post**, refer to the new value for the column **fname** as **post.fname**.

If the trigger event is a DELETE statement, using the new correlation name as a qualifier causes an error, because the column has no value after the row is deleted. For the rules that govern the use of correlation names, see Using Correlation Names in Triggered Actions on page 402.

You can use the REFERENCING clause for Delete only if you define a FOR EACH ROW trigger action.

REFERENCING Clause for Insert triggers

The REFERENCING clause for an Insert trigger can declare a correlation name for the inserted value in a column.

REFERENCING Clause for Insert triggers

" REFERENCINGNEW "

" [AS] "

" *correlation* "

Element	Description	Restrictions	Syntax
correlation	Name that you declare here to qualify a new column value (as *correlation*.*column*) in a triggered action	Must be unique within this CREATE TRIGGER statement	Identifier on page 1231

The *correlation* is a name for the new column value after the triggering statement has executed. Its scope of reference is only the FOR EACH ROW trigger action list; see Correlated Table Action on page 399. To use the correlation name, precede the column name with the *correlation* name, followed by a period (**.**) symbol. Thus, if the NEW *correlation* name is **post**, refer to the new value for the column **fname** as **post.fname**.

If the trigger event is an INSERT statement, using the old *correlation* name as a qualifier causes an error, because no value exists before the row is inserted. For the rules that govern how to use correlation names, see Using Correlation Names in Triggered Actions on page 402. You can use the INSERT REFERENCING clause only if you define a FOR EACH ROW trigger action.

The following example illustrates use of the INSERT REFERENCING clause. This example inserts a row into **backup_table1** for every row that is inserted into **table1**. The values that are inserted into **col1** and **col2** of **backup_table1** are an exact copy of the values that were just inserted into **table1**.

```
CREATE TABLE table1 (col1 INT, col2 INT);
CREATE TABLE backup_table1 (col1 INT, col2 INT);
CREATE TRIGGER before_trig
   INSERT ON table1    REFERENCING NEW AS new
   FOR EACH ROW
   (
   INSERT INTO backup_table1 (col1, col2)
   VALUES (new.col1, new.col2)
   );
```

As the preceding example shows, the REFERENCING clause for INSERT triggers allows you to refer to data values produced by the trigger action.

REFERENCING Clause for Update

The REFERENCING clause for an Update trigger can declare correlation names for the original value and for the updated value in a column.

(explicit id unique_526_Connect_42_rclu002) unique_526_Connect_42_rclu002

REFERENCING Clause for Update

" **REFERENCING** "

" { **OLD** | **NEW** } [**AS**] *correlation* "

Element	Description	Restrictions	Syntax
correlation	Name that you declare here to qualify an old or new column value (as *correlation*.*column*) in a triggered action	Must be unique within this CREATE TRIGGER statement	Identifier on page 1231

The OLD *correlation* is the name of the value of the column in the triggering table before execution of the triggering statement; the NEW *correlation* identifies the corresponding value after the triggering statement executes.

The scope of reference of the *correlation* names that you declare here is only within the FOR EACH ROW trigger action list. See Correlated Table Action on page 399.

To refer to an old or new column value, prefix the column name with the *correlation* name and a period (**.**) symbol. For example, if the new *correlation* name is **post**, you can refer to the new value in column **fname** as **post.fname**.

If the trigger event is an UPDATE statement, you can define both old and new *correlation* names to refer to column values before and after the triggering UPDATE statement. For rules that govern the use of *correlation* names, see Using Correlation Names in Triggered Actions on page 402.

You can use the UPDATE REFERENCING clause only if you define a FOR EACH ROW trigger action.

REFERENCING Clause for Select

The REFERENCING clause for a Select trigger can declare a correlation name for the value in a column.

REFERENCING Clause for Select

" **REFERENCING** "

" **OLD** "

" **[AS]** "

" *correlation* "

Element	Description	Restrictions	Syntax
correlation	Name that you declare here to qualify an old or new column value (as *correlation*.*column*) in a triggered action	Must be unique within this CREATE TRIGGER statement	Identifier on page 1231

This has the same syntax as the REFERENCING Clause for Delete on page 395. The scope of reference of the *correlation* name that you declare here is only within the FOR EACH ROW trigger action list. See Correlated Table Action on page 399.

You use the *correlation* name to refer to an old column value by preceding the column name with the *correlation* name and a period (**.**) symbol. For example, if the old *correlation* name is **pre**, you can refer to the old value for the column **fname** as **pre.fname**.

If the trigger event is a SELECT statement, using the new *correlation* name as a qualifier causes an error because the column does not have a new value after the column is selected. For the rules that govern the use of correlation names, see Using Correlation Names in Triggered Actions on page 402.

You can use the SELECT REFERENCING clause only if you define a FOR EACH ROW trigger action.

Correlated Table Action

Use the Correlated Trigger Action clause to define the SQL statements that are executed as the trigger action when a triggering event activates a trigger on a table.

(explicit id unique_528_Connect_42_corta001) unique_528_Connect_42_corta001

Correlated Table Action

" [**BEFORE** <Triggered-Action>] "

" **FOR EACH ROW** <Triggered-Action> "

" [**AFTER** <Triggered-Action>] "

If the CREATE TRIGGER statement contains an INSERT REFERENCING clause, a DELETE REFERENCING clause, an UPDATE REFERENCING clause, or a SELECT REFERENCING clause, you must include a FOR EACH ROW triggered-action list in the Action clause. You can also include BEFORE and AFTER triggered-action lists, but they are optional.

For information on the BEFORE, FOR EACH ROW, and AFTER triggered-action lists, see Action Clause on page 392

Triggered Action on a Table

The Triggered Action specifies a list of SQL statements to execute when a trigger is activated. The BEFORE, FOR EACH ROW, and AFTER sections of the Action Clause can each specify different list of triggered actions for the same trigger.

(explicit id unique_529_Connect_42_tral003) unique_529_Connect_42_tral003(explicit id unique_529_Connect_42_tral004)
unique_529_Connect_42_tral004(explicit id unique_529_Connect_42_tral005) unique_529_Connect_42_tral005(explicit id unique_529_Connect_42_tral006)
unique_529_Connect_42_tral006(explicit id unique_529_Connect_42_tral007) unique_529_Connect_42_tral007(explicit id unique_529_Connect_42_tral008)
unique_529_Connect_42_tral008

Triggered Action

" [**WHEN** (<Condition>)] ({ | <INSERT Statement> | <DELETE Statement> | <UPDATE Statement> | <EXECUTE PROCEDURE Statement> | <EXECUTE FUNCTION Statement> }) "

For a trigger on a table, the trigger action consists of an optional WHEN condition and the action statements. You can specify a triggered-action list for each WHEN clause, or you can specify a single list (of one or more trigger actions) if you include no WHEN clause.

Database objects that are referenced explicitly in the trigger action or in the definition of the trigger event, such as tables, columns, and UDRs, must exist when the CREATE TRIGGER statement defines the new trigger.

The following Update trigger includes no WHEN clause, but invokes the SPL routine **upd_items_p1( )** as its triggered action whenever the **quantity** column of the **items** table is updated:

```
CREATE TRIGGER up_itemqty
   UPDATE OF quantity ON items
   BEFORE(EXECUTE PROCEDURE upd_items_p1);
```

If no **upd_items_p1( )** routine is registered in the database when this CREATE TRIGGER statement is issued, the database server issues an error, and no **up_itemqty** trigger is created.

Attention: When you specify a date expression in the WHEN condition or in an action statement, make sure to specify four digits instead of two digits for the year. For more about abbreviated years, see the description of **DBCENTURY** in the *HCL® Informix® Guide to SQL: Reference*, which also describes how the behavior of some database objects can be affected by environment variable settings. Like fragmentation expressions, check constraints, and UDRs, triggers are stored in the system catalog with the creation-time settings of environment variables that can affect the evaluation of expressions like the WHEN condition. The database server ignores any subsequent changes to those settings when evaluating expressions in those database objects.

WHEN Condition

The WHEN condition makes the triggered action dependent on the outcome of a test. When you include a WHEN condition in a triggered action, the statements in the triggered action list execute only if the condition evaluates to `true`. If the WHEN condition evaluates to `false` or `unknown`, then the statements in the triggered action list are not executed.

If the triggered action is in a FOR EACH ROW section, its condition is evaluated for each row. For example, the triggered action in the following trigger executes only if the condition in the WHEN clause is true:

```
CREATE TRIGGER up_price
   UPDATE OF unit_price ON stock
   REFERENCING OLD AS pre NEW AS post
   FOR EACH ROW WHEN(post.unit_price > pre.unit_price * 2)
   (INSERT INTO warn_tab VALUES(pre.stock_num, pre.order_num,
      pre.unit_price, post.unit_price, CURRENT));
```

An SPL routine that executes inside the WHEN condition carries the same restrictions as a UDR that is called in a data manipulation statement. That is, the SPL routine cannot contain certain SQL statements. For information on which statements are restricted, see Restrictions on SPL Routines in Data-Manipulation Statements on page 1293.

Action Statements

The triggered-action statements can be INSERT, DELETE, UPDATE, EXECUTE FUNCTION, or EXECUTE PROCEDURE statements. If the action list contains multiple statements, and the WHEN condition is satisfied (or is absent), then these statements execute in the order in which they appear in the list.

UDRs as Triggered Actions

Calls to user-defined functions and procedures, including trigger routines, can be triggered actions. The triggered action list of the FOR EACH ROW clause can include calls to UDRs that call **mi_trigger*( )** functions. Triggered actions are the only context in which a trigger routine of Informix® can be invoked. For restrictions on the calling context and the syntax of trigger routines, see The REFERENCING and FOR Clauses on page 230.

You can use the EXECUTE FUNCTION statement to call any user-defined function or trigger function. Use the EXECUTE PROCEDURE statement to call any user-defined procedure or trigger procedure.

In contexts where Boolean expressions are valid, the Boolean operators SELECTING, INSERTING, DELETING, and UPDATING are valid in trigger routines and in other UDRs that are invoked in triggered action statements. These operators return TRUE ('t') if the triggering event matches the DML operation that matches the name of the operator; otherwise they return FALSE ('f'). A single trigger routine can be designed to perform different triggered actions for different types of triggering events, using these Boolean operators to execute program blocks that are appropriate to the type of trigger.

For restrictions on using SPL routines as triggered actions, see Rules for SPL Routines on page 407 and Triggers and SPL Routines on page 384.

Achieving a Consistent Result

To guarantee that the triggering statement returns the same result with and without the triggered actions, make sure that the triggered actions in the BEFORE and FOR EACH ROW sections do not modify any table referenced in the following clauses:

- WHERE clause
- SET clause in the UPDATE statement
- SELECT clause
- EXECUTE PROCEDURE clause or EXECUTE FUNCTION clause in a multiple-row INSERT statement.

Declaring keywords of SQL as correlation names

If you use the INSERT, DELETE, UPDATE, or EXECUTE keywords as a *correlation* identifier in any of the following clauses inside a triggered action list, you must qualify them by the *owner* name, the *table* name, or both:

- FROM clause of a SELECT statement
- INTO clause of the EXECUTE PROCEDURE or EXECUTE FUNCTION statement
- GROUP BY clause
- SET clause of the UPDATE statement.

The database server issues a syntax error if these keywords are *not* qualified when you include these clauses inside a triggered action.

If you use the keyword as a column name, it must be qualified by the table name; for example, **table.update**. If both the table name and the column name are keywords, they must be qualified by the owner name (for example, *owner***.insert.update**). If the owner name, table name, and column name are all keywords, the owner name must be in quotation marks; for example,

'delete'.insert.update. (These are general rules regarding reserved words as identifiers, rather than special cases for triggers. Your code will be easier to read and to maintain if you avoid using the keywords of SQL as identifiers.)

The only exception is when these keywords are the first table or column name in the list, and you do not need to qualify them. For example, **delete** in the following statement does not need to be qualified because it is the first column listed in the INTO clause:

```
CREATE TRIGGER t1 UPDATE OF b ON tab1
   FOR EACH ROW (EXECUTE PROCEDURE p2() INTO delete, d);
```

The following statements show examples in which you must qualify the column name or the table name:

- FROM clause of a SELECT statement

  ```
  CREATE TRIGGER t1 INSERT ON tab1
     BEFORE (INSERT INTO tab2 SELECT * FROM tab3, 'owner1'.update);
  ```

- INTO clause of an EXECUTE PROCEDURE statement

  ```
  CREATE TRIGGER t3 UPDATE OF b ON tab1
     FOR EACH ROW (EXECUTE PROCEDURE p2() INTO
     d, tab1.delete);
  ```

  An INSTEAD OF trigger on a view cannot include the EXECUTE PROCEDURE INTO statement among its triggered actions.

- GROUP BY clause of a SELECT statement

  ```
  CREATE TRIGGER t4 DELETE ON tab1
     BEFORE (INSERT INTO tab3 SELECT deptno, SUM(exp)
     FROM budget GROUP BY deptno, budget.update);
  ```

- SET clause of an UPDATE statement

  ```
  CREATE TRIGGER t2 UPDATE OF a ON tab1
     BEFORE (UPDATE tab2 SET a = 10, tab2.insert = 5);
  ```

Using Correlation Names in Triggered Actions

These rules apply when you use correlation names in triggered actions:

- You can use the correlation names as qualifiers for the old and new column values in SQL statements of the FOR EACH ROW triggered-action list and in the WHEN condition.
- The WHEN conditions and FOR EACH ROW clauses of multiple triggers on the same table can use different correlated variables in the REFERENCING clauses of triggers and of trigger routines to reference values in the same column.
- The old and new correlation names refer to all rows affected by the triggering statement.
- You cannot use the correlation name to qualify a column name in the GROUP BY, the SET, or the COUNT DISTINCT clause.

- The scope of reference of the correlation names is the entire trigger definition. This scope is statically determined, meaning that it is limited to the trigger definition; it does not encompass cascading triggers or columns that are qualified by a table name in a UDR that is a triggered action, except for trigger routines that are invoked in the FOR EACH ROW clause.

For additional information on using correlation names in trigger routines, see Rules for SPL Routines on page 407.

When to Use Correlation Names

In SQL statements of the FOR EACH ROW list, you must qualify all references to columns in the triggering table with either the old or new correlation name, unless the statement is valid independent of the triggered action.

In other words, if a column name inside a FOR EACH ROW triggered action list is not qualified by a correlation name, even if it is qualified by the triggering table name, it is interpreted as if the statement were independent of the triggered action. No special effort is made to search the definition of the triggering table for the non-qualified column name.

For example, assume that the following DELETE statement is a triggered action inside the FOR EACH ROW section of a trigger:

```
DELETE FROM tab1 WHERE col_c = col_c2;
```

For the statement to be valid, both **col_c** and **col_c2** must be columns from **tab1**. If **col_c2** is intended to be a correlation reference to a column in the triggering table, it must be qualified by either the old or the new correlation name. If **col_c2** is not a column in **tab1** and is not qualified by either the old or new correlation name, you get an error.

In a statement that is valid independent of the triggered action, a column name with no *correlation* qualifier refers to the current value in the database.

In the triggered action for trigger **t1** in the next example, **mgr** in the WHERE clause of the correlated subquery is an unqualified column in the triggering table. In this case, **mgr** refers to the current column value in **empsal** because the INSERT statement is valid independent of the triggered action.

```
CREATE DATABASE db1;
CREATE TABLE empsal (empno INT, salary INT, mgr INT);
CREATE TABLE mgr (eno INT, bonus INT);
CREATE TABLE biggap (empno INT, salary INT, mgr INT);

CREATE TRIGGER t1 UPDATE OF salary ON empsal
AFTER (INSERT INTO biggap SELECT * FROM empsal WHERE salary <
   (SELECT bonus FROM mgr WHERE eno = mgr));
```

In a triggered action, an unqualified column name from the triggering table refers to the current column value, but only when the triggered statement is valid independent of the triggered action.

Qualified Versus Unqualified Value

The following table summarizes what value is retrieved when the **column** name is qualified by the *old* or by the *new* correlation name after various trigger events.

Trigger Event	*old*.column	*new*.column
INSERT	No value (error)	Inserted value
UPDATE (column updated)	Original value	Current® value (U)
UPDATE (column not updated)	Original value	Original value (N)
DELETE	Original value	No value (error)
SELECT	Original value	No value (error)

When a correlation name has no value, an error is issued only when an SQL or SPL statement referencing the undefined correlation is executed, rather than when the correlation name is declared. Refer to the following key when you read the previous table.

Term

Meaning

Original value

Value before the triggering event

Current® value

Value after the triggering event

(N)

Cannot be changed by triggered action

(U)

Can be updated by triggered actions; updated value might be different from the original value because of preceding triggered actions.

Outside a FOR EACH ROW triggered-action list, you cannot qualify a column from the triggering table with either the old correlation name or the new correlation name; it always refers to the current value in the database.

Statements in the trigger action list use whatever collating order was in effect when the trigger was created, even if a different collation is in effect when the trigger action is executed. See SET COLLATION statement on page 775 for details of how to specify a collating order different from what **DB_LOCALE** specifies.

Re-Entrancy of Triggers

In some cases a trigger can be re-entrant. In these cases the triggered action can reference the triggering table. In other words, both the trigger event and the triggered action can operate on the same table. The following list summarizes the situations in which triggers can be re-entrant and the situations in which triggers cannot be re-entrant:

- The trigger action of an Update trigger cannot be an INSERT or DELETE of the table that the trigger event updated.
- Similarly, the trigger action of an Update trigger cannot be an UPDATE of a column that the trigger event updated. (But the trigger action of an Update trigger can update a column that was not updated by the trigger event.)

For example, assume that the following UPDATE statement, which updates columns **a** and **b** of **tab1**, is the triggering statement:

```
UPDATE tab1 SET (a, b) = (a + 1, b + 1);
```

Now consider the trigger actions in the following example. The first UPDATE statement is a valid trigger action, but the second one is not, because it updates column **b** again.

```
UPDATE tab1 SET c = c + 1;    -- OK
UPDATE tab1 SET b = b + 1;    -- INVALID
```

- If the trigger has an UPDATE event, the trigger action can be an EXECUTE PROCEDURE or EXECUTE FUNCTION statement with an INTO clause that references a column that was updated by the trigger event or any other column in the triggering table.

  When an EXECUTE PROCEDURE or EXECUTE FUNCTION statement is the trigger action, the INTO clause for an UPDATE trigger is valid only in FOR EACH ROW trigger actions, and column names that appear in the INTO clause must be from the triggering table.

  The following statement illustrates the appropriate use of the INTO clause:

  ```
  CREATE TRIGGER upd_totpr UPDATE OF quantity ON items
     REFERENCING OLD AS pre_upd NEW AS post_upd
     FOR EACH ROW(EXECUTE PROCEDURE
        calc_totpr(pre_upd.quantity,post_upd.quantity,
        pre_upd.total_price) INTO total_price);
  ```

  The column that follows the INTO keyword must be in the triggering table, but need not have been updated by the trigger event.

  When the INTO clause appears in the EXECUTE PROCEDURE or EXECUTE FUNCTION statement, the database server updates the specified columns with values returned from the UDR, immediately upon returning from the UDR.
- If the trigger has an INSERT event, the trigger action cannot be an INSERT or DELETE statement that references the triggering table.
- If the trigger has an INSERT event, the trigger action can be an UPDATE statement that references a column in the triggering table, but this column cannot be a column for which a value was supplied by the trigger event.

  If the trigger has an INSERT event, and the trigger action updates the triggering table, the columns in both statements must be mutually exclusive. For example, assume that the triggering statement inserts values for columns **cola** and **colb** of table **tab1**:

  ```
  INSERT INTO tab1 (cola, colb) VALUES (1,10);
  ```

  Now consider the following trigger actions. The first UPDATE is valid, but the second one is not, because it updates column **colb** even though the trigger event already supplied a value for column **colb**:

  ```
  UPDATE tab1 SET colc=100; --OK
  UPDATE tab1 SET colb=100; --INVALID
  ```

- If the trigger has an INSERT event, the trigger action can be an EXECUTE PROCEDURE or EXECUTE FUNCTION statement with an INTO clause that references a column that was supplied by the trigger event or a column that was not supplied by the trigger event.

  When an EXECUTE PROCEDURE or EXECUTE FUNCTION statement is the trigger action, you can specify the INTO clause for an INSERT trigger only when the trigger action occurs in the FOR EACH ROW list. In this case, the INTO clause can contain only column names from the triggering table.

  The following statement illustrates the valid use of the INTO clause:

  ```
  CREATE TRIGGER ins_totpr INSERT ON items
     REFERENCING NEW AS new_ins
     FOR EACH ROW (EXECUTE PROCEDURE calc_totpr
        (0, new_ins.quantity, 0) INTO total_price);
  ```

  The column that follows the INTO keyword can be a column in the triggering table that was supplied by the trigger event, or a column in the triggering table that was not supplied by the trigger event.

  When the INTO clause appears in the EXECUTE PROCEDURE or the EXECUTE FUNCTION statement, the database server immediately updates the specified columns with values returned from the UDR.

- If the trigger action is a SELECT statement, the SELECT statement can reference the triggering table. The SELECT statement can be a trigger action in the following instances:
  - The SELECT statement appears in a subquery in the WHEN clause or in a trigger-action statement.
  - The trigger action is a UDR, and the SELECT statement appears inside the UDR.

Re-Entrancy and Cascading Triggers

The cases when a trigger cannot be re-entrant apply recursively to all cascading triggers, which are considered part of the initial trigger. In particular, this rule means that a cascading trigger cannot update any columns in the triggering table that were updated by the original triggering statement, including any nontriggering columns affected by that statement. For example, assume this UPDATE statement is the triggering statement:

```
UPDATE tab1 SET (a, b) = (a + 1, b + 1);
```

In the cascading triggers of the next example, **trig2** fails at runtime because it references column **b**, which the triggering UPDATE statement updates:

```
CREATE TRIGGER trig1 UPDATE OF a ON tab1-- Valid
   AFTER (UPDATE tab2 SET e = e + 1);

CREATE TRIGGER trig2 UPDATE OF e ON tab2-- Invalid
   AFTER (UPDATE tab1 SET b = b + 1);
```

Now consider the following SQL statements. When the final UPDATE statement is executed, column **a** is updated and the trigger **trig1** is activated.

The trigger action again updates column **a** with an EXECUTE PROCEDURE INTO statement.

```
CREATE TABLE temp1 (a INT, b INT, e INT);
INSERT INTO temp1 VALUES (10, 20, 30);
```

```
CREATE PROCEDURE proc(val iINT) RETURNING INT,INT;
    RETURN val+10, val+20;
END PROCEDURE;

CREATE TRIGGER trig1 UPDATE OF a ON temp1
    FOR EACH ROW (EXECUTE PROCEDURE proc(50) INTO a, e);

CREATE TRIGGER trig2 UPDATE OF e ON temp1
    FOR EACH ROW (EXECUTE PROCEDURE proc(100) INTO a, e);

UPDATE temp1 SET (a,b) = (40,50);
```

Several questions arise from this example of cascading triggers. First, should the update of column **a** activate trigger **trig1** again? The answer is no. Because the trigger was activated, it is not activated a second time. If the trigger action is an EXECUTE PROCEDURE INTO or EXECUTE FUNCTION INTO statement, the only triggers that are activated are those that are defined on columns that are mutually exclusive from the columns updated until then (in the cascade of triggers) in that table. Other triggers are ignored.

Another question that arises from the example is whether trigger **trig2** should be activated. The answer is yes. The trigger **trig2** is defined on column **e.** Until now, column **e** in table **temp1** has not been modified. Trigger **trig2** is activated.

A final question that arises from the example is whether triggers **trig1** and **trig2** should be activated after the trigger action in **trig2** is performed. The answer is no. Neither trigger is activated. By this time columns **a** and **e** have been updated once, and triggers **trig1** and **trig2** have been executed once. The database server ignores and does not activate these triggers. For more about cascading triggers, see Cascading Triggers on page 410.

As noted earlier, an INSTEAD OF trigger on a view cannot include the EXECUTE PROCEDURE INTO statement among its trigger actions. In addition, an error results if two views each have INSERT INSTEAD OF triggers with actions defined to perform INSERT operations on the other view.

Rules for SPL Routines

In addition to the rules listed in Re-Entrancy of Triggers on page 404, the following guidelines apply to an SPL routine that is specified as a trigger action:

- The SPL routine cannot be a cursor function (one that returns more than one row) in a context where only one row is expected.
- You cannot use the old or new correlation name inside the SPL routine unless the CREATE FUNCTION or CREATE PROCEDURE statement includes the REFERENCING clause that defines the UDR as a trigger routine. If you need to use the corresponding values in a routine that is not a trigger routine, you must pass them as parameters. In this case, the routine should be independent of triggers, and the old or new correlation name does not have any meaning outside the trigger.
- A trigger routine must include the REFERENCING clause that can declare a correlation name for OLD or NEW column values that SPL statements in the trigger routine can reference.

- A trigger routine must include the FOR *table_object* clause that specifies the name of the table or view in the local database whose triggers can invoke this routine. The triggered action cannot call a trigger routine that does not specify the triggering table or view.
- Only trigger routines invoked in the FOR EACH ROW section of the Triggered Action list can operate directly on old or new correlation names that are defined in the REFERENCING clause of the trigger or of the trigger routine.
- Trigger routines can be invoked only in the FOR EACH ROW section of the Triggered Action list in the trigger definition.
- Correlated variables for OLD or NEW values can appear in the IF statement of SPL and in CASE expressions.
- Only correlated variables for NEW values can be on the left-hand side of a LET expression that references correlated variables. In this case, the FOR clause of the SPL routine must specify a table, rather than a view, and the trigger whose action invokes the SPL routine cannot be an INSTEAD OF trigger.
- Both OLD and NEW values can be on the right-hand side of a LET expression.
- Only trigger routines that are invoked in the FOR EACH ROW clause can use the Boolean operators SELECTING, INSERTING, DELETING, and UPDATING. These operators return TRUE ('t') if the triggering event matches the DML operation referenced by the name of the operator, and they return FALSE ('f') otherwise.
- The IF statement of SPL and CASE expressions of SQL can specify these operators as the condition in a trigger routine.
- Trigger routines must be written in the SPL language. They cannot be written in an external language, such as the C or Java™ language, but the trigger routine can include calls to external language routines, such as the **mi_trigger** application programming interface for trigger introspection.
- Trigger routines cannot reference savepoints. Any changes to the data values or to the schema of the database by a triggered action must be committed or rolled back in their entirety. Informix® does not support the ROLLBACK TO SAVEPOINT statement in a trigger routine for the partial rollback of a triggered action.

For more information about the **mi_trigger** API, refer to the *HCL® Informix® DataBlade® API Programmer's Guide* and to the *IBM® Informix® DataBlade® API Function Reference*.

When you use an SPL routine as a trigger action, the database objects that the routine references are not checked until the routine is executed.

See also the SPL restrictions in Triggers and SPL Routines on page 384.

Privileges to Execute Trigger Actions

If you do not own the trigger, but the access privileges held by the trigger owner include WITH GRANT OPTION, then for each triggered SQL statement you inherit the privileges of the trigger owner (with grant option), in addition to any privileges that have been granted to you individually, or through an active or default role that you hold, or that you hold as a member of the PUBLIC group. If the triggered action calls a UDR, you need Execute privilege on the UDR, or the trigger owner must have Execute privilege with grant option.

Important: As a security precaution, discretionary access privileges that the user holds only from a role (but that were not granted to the user individually or as member of the PUBLIC group) cannot provide access to tables outside the current database through a triggered action or through a trigger routine.

As a security precaution, however, discretionary access privileges that the user holds only from a role (but that were not granted to the user individually or as member of the PUBLIC group) cannot provide access to tables outside the current database through a triggered action or through a trigger routine.

While executing the UDR, however, you do not inherit the privileges of the trigger owner; instead, you receive the privileges granted with the UDR, depending on whether the routine is a DBA-privileged or an owner-privileged UDR:

1. Privileges for a DBA-privileged UDR

   When a UDR is registered with the DBA keyword, and you are granted the Execute privilege on the UDR, the database server automatically grants you temporary DBA privileges that are available only when you are executing the UDR.

2. Privileges for an owner-privileged UDR

   If the UDR was created without the DBA keyword, but the owner of the UDR was granted the necessary privileges on the underlying database objects with the WITH GRANT OPTION keywords, then you inherit these privileges when you are granted the Execute privilege on the UDR.

For a UDR that is not DBA privileged, all non-qualified database objects that the UDR references are implicitly qualified by the name of the UDR owner.

If the UDR owner has no WITH GRANT OPTION privilege, you have your original privileges on the underlying database objects when the UDR executes. For more information on privileges on SPL routines, refer to the *HCL® Informix® Guide to SQL: Tutorial*.

A view that has no INSTEAD OF trigger has only Select (with grant option) privilege. If an INSTEAD OF trigger is created on it, however, then the view has Insert (with grant option) privilege during creation of the trigger. The view owner can now grant only Select and Insert privileges to others. This is independent of the trigger action. It is not necessary to obtain Execute (with grant option) privilege on the procedure or function. By default, Execute privilege (without grant option) is granted on each UDR in the action list.

You can use roles with triggers. Role-related statements (CREATE ROLE, DROP ROLE, GRANT, REVOKE, and SET ROLE) and SET SESSION AUTHORIZATION statements are valid in a UDR that the triggered action invokes. Privileges that a user acquired by enabling a role or by a SET SESSION AUTHORIZATION statement are not relinquished when a trigger is executed.

On a complex view (one with columns from more than one table), only the owner or DBA can create an INSTEAD OF trigger. The owner receives Select privileges when the trigger is created. Only after obtaining the required Execute privileges can the owner of the view grant privileges to other users. When the trigger on the complex view is dropped, all of these privileges are revoked.

Creating a Trigger Action That Anyone Can Use

About this task

For a trigger to be executable by anyone who has the privileges to execute the triggering statement, you can ask the DBA to create a DBA-privileged UDR and grant you the Execute privilege with the WITH GRANT OPTION right.

You then use the DBA-privileged UDR as the trigger action. Anyone can execute the trigger action because the DBA-privileged UDR carries the WITH GRANT OPTION right. When you activate the UDR, the database server applies privilege-checking rules for a DBA.

Cascading Triggers

The database server allows triggers other than Select triggers to cascade, meaning that the trigger actions of one trigger can activate another trigger. (For further information on the restriction against cascading Select triggers, see Circumstances When a Select Trigger Is Activated on page 389.)

The maximum number of triggers in a cascading series is 61: the initial trigger plus a maximum of 60 cascading triggers. When the number of cascading triggers in a series exceeds the maximum, the database server returns error number -748, with the following message:

```
Exceeded limit on maximum number of cascaded triggers.
```

The next example illustrates a series of cascading triggers that enforce referential integrity on the **manufact**, **stock**, and **items** tables in the **stores_demo** database. When a manufacturer is deleted from the **manufact** table, the first trigger, **del_manu**, deletes all the items of that manufacturer from the **stock** table. Each DELETE in the **stock** table activates a second trigger, **del_items**, that deletes all **items** of that manufacturer from the **items** table. Finally, each DELETE in the **items** table triggers SPL routine **log_order**, creating a record of any orders in the **orders** table that can no longer be filled.

```
CREATE TRIGGER del_manu
   DELETE ON manufact REFERENCING OLD AS pre_del
   FOR EACH ROW(DELETE FROM stock WHERE manu_code = pre_del.manu_code);
CREATE TRIGGER del_stock
   DELETE ON stock REFERENCING OLD AS pre_del
   FOR EACH ROW(DELETE FROM items WHERE manu_code = pre_del.manu_code);
CREATE TRIGGER del_items
   DELETE ON items REFERENCING OLD AS pre_del
   FOR EACH ROW(EXECUTE PROCEDURE log_order(pre_del.order_num));
```

When you are not using logging, referential integrity constraints on both the **manufact** and **stock** tables prohibit the triggers in this example from executing. When you use logging, however, the triggers execute successfully because constraint checking is deferred until all the trigger actions are complete, including the actions of cascading triggers. For more information about how constraints are handled when triggers execute, see Constraint Checking on page 411.

The database server prevents loops of cascading triggers by not allowing you to modify the triggering table in any cascading trigger action, except with an UPDATE statement that does not modify any column that the triggering UPDATE statement updated, or with an INSERT statement. An INSERT trigger can define UPDATE trigger actions on the same table.

Constraint Checking

When you use logging, the database server defers constraint checking on the triggering statement until after the statements in the triggered-action list execute. This is equivalent to executing a SET CONSTRAINTS ALL DEFERRED statement before executing the triggering statement. After the trigger action is completed, the database server effectively executes a SET CONSTRAINTS *constraint* IMMEDIATE statement to check the constraints that were deferred. This action allows you to write triggers so that the trigger action can resolve any constraint violations that the triggering statement creates. For more information, see SET Database Object Mode statement on page 784.

Consider the following example, in which the table **child** has constraint **r1**, which references the table **parent**. You define trigger **trig1** and activate it with an INSERT statement. In the trigger action, **trig1** checks to see if **parent** has a row with the value of the current **cola** in **child**; if not, it inserts it.

```
CREATE TABLE parent (cola INT PRIMARY KEY);
CREATE TABLE child (cola INT REFERENCES parent CONSTRAINT r1);
CREATE TRIGGER trig1 INSERT ON child
   REFERENCING NEW AS new
   FOR EACH ROW
   WHEN((SELECT COUNT (*) FROM parent
      WHERE cola = new.cola) = 0)
-- parent row does not exist
   (INSERT INTO parent VALUES (new.cola));
```

When you insert a row into a table that is the child table in a referential constraint, the row might not exist in the parent table. The database server does not immediately return this error on a triggering statement. Instead, it allows the trigger action to resolve the constraint violation by inserting the corresponding row into the parent table. As the previous example shows, you can check within the trigger action to see whether the parent row exists, and if so, you can provide logic to bypass the INSERT action.

For a database without logging, the database server does not defer constraint checking on the triggering statement. In this case, the database server immediately returns an error if the triggering statement violates a constraint.

You cannot use the SET Transaction Mode statement in a trigger action. The database server checks this restriction when you activate a trigger, because the statement could occur inside a UDR.

Preventing Triggers from Overriding Each Other

When you activate multiple triggers with an UPDATE statement, a trigger can possibly override the changes that an earlier trigger made. If you do not want the trigger actions to interact, you can split the UPDATE statement into multiple UPDATE statements, each of which updates an individual column.

As another alternative, you can create a single update trigger for all columns that require a trigger action. Then, inside the trigger action, you can test for the column being updated and apply the actions in the desired order. This approach, however, is different from having the database server apply the actions of individual triggers, and it has the following disadvantages:

- If the triggering UPDATE statement sets a column to the current value, you cannot detect the UPDATE, so the trigger action is skipped. You might wish to execute the trigger action, even though the value of the column has not changed.
- If the trigger has a BEFORE action, it applies to all columns, because you cannot yet detect whether a column has changed.

Tables in Remote Databases

You cannot create triggers on tables or views that reside outside the current database. You can, however, define a trigger on a local table whose trigger action manipulates a table in another database of the local server instance, or a table in a database of another server instance.

The following example defines an Update trigger on the **newtab** table in the current database of the local Informix® server instance, **dbserver1**, to which the session is connected. Here the trigger action specifies an UPDATE operation on the **items** table of the **stores_demo** database of the remote **dbserver2** Informix® server instance:

```
CREATE TRIGGER upd_nt UPDATE ON newtab
   REFERENCING NEW AS post
   FOR EACH ROW(UPDATE stores_demo@dbserver2:items
      SET quantity = post.qty WHERE stock_num = post.stock
      AND manu_code = post.mc);
```

In summary, triggers registered in the local database can support local, cross-database, and cross-server trigger actions:

- local trigger actions on a table in the local database
- cross-database trigger actions on a table in another database of the local server instance
- cross-server trigger actions on a table in a database of a remote server instance.

The cross-server triggered action of a trigger that is defined in a database of a remote server instance can be the event that activates one or more triggers in the local database, but in this case, triggered actions of the local trigger cannot be cross-server operations. If a SELECT, DELETE, INSERT, MERGE, or UPDATE statement from a remote database server is the event that activates a local trigger whose action specifies a table in a database of a remote server instance, the trigger actions fail.

For example, the following combination of trigger action and triggering statement results in an error when the triggering statement executes:

```
-- Trigger action from dbserver1 to dbserver3:
CREATE TRIGGER upd_nt UPDATE ON newtab
   REFERENCING NEW AS post
   FOR EACH ROW(UPDATE stores_demo@dbserver3:items
      SET quantity = post.qty WHERE stock_num = post.stock
      AND manu_code = post.mc);

-- Triggering statement from dbserver2:
UPDATE stores_demo@dbserver1:newtab
   SET qty = qty * 2 WHERE s_num = 5
   AND mc = 'ANZ';
```

The UPDATE statement above returns an error at run time, because a cross-server triggering event cannot trigger another cross-server action.

Important: As a security precaution, discretionary access privileges that a user holds only from a role cannot provide access to tables outside the current database through a view or through a trigger. Cross-database trigger actions and cross-server trigger actions require access privileges on the non-local database and table that were granted directly to the user, or granted to the PUBLIC group.

Logging and Recovery

You can create triggers for databases, with and without logging. If the trigger fails in a database that has transaction logging, the triggering statement and trigger actions are rolled back, as if the actions were an extension of the triggering statement, but the rest of the transaction is not rolled back.

In a database that does not have transaction logging, however, you cannot roll back when the triggering statement fails. In this case, you are responsible for maintaining data integrity in the database. The UPDATE, INSERT, or DELETE action of the triggering statement occurs before the trigger actions in the FOR EACH ROW section. If the trigger action fails for a database without logging, the application must restore the row that was changed by the triggering statement to its previous value.

If a trigger action calls a UDR, but the UDR terminates in an exception-handling section, any actions that modify data inside that section are rolled back with the triggering statement. In the following partial example, when the exception handler traps an error, it inserts a row into the table **logtab**:

```
ON EXCEPTION IN (-201)
   INSERT INTO logtab values (errno, errstr);
   RAISE EXCEPTION -201
END EXCEPTION;
```

When the RAISE EXCEPTION statement returns the error, however, the database server rolls back this INSERT because it is part of the trigger actions. If the UDR is executed outside a trigger action, the INSERT is not rolled back.

The UDR that implements a trigger action cannot contain any BEGIN WORK, COMMIT WORK, or ROLLBACK WORK statements. If the database has transaction logging, you must either begin an explicit transaction before the triggering statement, or the statement itself must be an implicit transaction. In any case, no other transaction-related statement is valid inside the UDR.

You can use triggers to enforce referential actions that the database server does not currently support. In a database without logging, you are responsible for maintaining data integrity when the triggering statement fails.

CREATE TRUSTED CONTEXT statement

Use the CREATE TRUSTED CONTEXT statement to define a newtrusted-context object. This statement is an extension to the ANSI/ISO standard for the SQL language.

You must hold the database security administrator (DBSECADM) role to run this statement.

Syntax

(explicit id unique_576_Connect_42_ctcx001) unique_576_Connect_42_ctcx001 **CREATE TRUSTED CONTEXT***context* { **USER** | **BASED UPON CONNECTION USING SYSTEM AUTHID** } *user* { | [ ] **ATTRIBUTES** ({ **ADDRESS** '*address* ' }) | [[ ] **WITH USE FOR** <Authorized User Clause>] | [{ [ ] | **NO DEFAULT ROLE** | **DEFAULT ROLE***role* }] | [{ [ ] | **DISABLE** | **ENABLE** }] }

Authorized User Clause

" { *user* [**ROLE***role*] | **PUBLIC** } [{ **WITHOUT AUTHENTICATION** | **WITH AUTHENTICATION** }] "

Element	Description	Restrictions	Syntax
address	Communication address of the client connection to the database server	Must be unique among communication addresses of clients for this trusted-context object. For additional *address* restrictions, see **ADDRESS attributes** below.	Quoted String on page 1202
context	Name declared here for the trusted-context object	Must be unique among the names of trusted-context objects of this database server instance, and cannot begin with the characters SYS	Identifier on page 1231
role	An existing user-defined or built-in role	Must exist in the database, and must be unique among attributes of this trusted-context object	Owner name on page 1262
user	Authorization identifier of a user	Must be a valid authorization identifier. Cannot be longer than 32 bytes. Must not be the authorization ID of the user who issues this statement. Must not be specified more than once in the WITH USE FOR clause.	Owner name on page 1262

Usage

The CREATE TRUSTED CONTEXT statement is used to create trusted-context objects, which can allow users to have trusted connections. Within the CREATE TRUSTED CONTEXT STATEMENT, each ATTRIBUTES, DEFAULT ROLE, ENABLE, and WITH USE clause can be specified no more than once, and each attribute name and corresponding value must be unique.

USER clause

The USER clause specifies the system authorization ID that can establish the context created in this SQL statement.

ADDRESS attributes

The ATTRIBUTES clause can specify one or more communication addresses for connections to the database server on which the trusted-context object is defined. The following restrictions apply to communication addresses that the ALTER TRUSTED CONTEXT or the CREATE TRUSTED CONTEXT statements reference:

- Each must be unique among communication addresses of clients for this trusted-context object.
- Each must conform to the TCP/IP protocol.
- Each must be an IPv4 address, an IPv6 address, or a secure domain name.
- An IPv4 address or IPv6 address must be a real host address (not a local host), and must not contain leading blank spaces.
- An IPv6 address, in addition, must not be an IPv4-mapped IPv6 address.
- A secure domain name must not be a Dynamic Host Configuration Protocol (DHCP) address.

If an *address* value is the name of a secure domain, that name is converted to an IP address by the domain-name server, where a resulting IPv4 or IPv6 address is determined. When a domain name is converted to an IP address, the result of this conversion might be a set of one or more IP addresses. In this case, the database server interprets an incoming connection request as matching the ADDRESS attribute of a trusted-context object if the IP address from which the connection originates matches any of the IP addresses to which the domain name was converted.

The ADDRESS attribute can be specified multiple times, but each *address* pair must be unique for the set of attributes.

Attention:

If you have an existing application that includes the ENCRYPTION or WITH ENCRYPTION options in the ATTRIBUTES clause, you can leave them without the database server issuing an SQL error. Except for `WITH ENCRYPTION 'NONE'` and `ENCRYPTION 'NONE'`, however, these encryption options of the CREATE TRUSTED CONTEXT statement are not supported for Informix® database servers.

WITH USE FOR clause

The WITH USE FOR clause specifies that the trusted connection can be used by the specified authorization identifier. The same *user* name cannot appear more than once in this clause, which allows access by both the list of specified users and by PUBLIC.

For example, assume that a trusted-context object is defined that allows access by both `PUBLIC WITH AUTHENTICATION` and `joe WITHOUT AUTHENTICATION`. If the trusted-context object is used by `joe`, authentication is not required. If the trusted-context object is used by `george`, however, who has access only as a member of `PUBLIC`, authentication is required.

The WITH AUTHENTICATION attribute specifies that switching the current user on a trusted connection based on this trusted-context object to this user requires authentication. The WITHOUT AUTHENTICATION attribute specifies that switching the current user does not require authentication. The specifications for a user override the specifications for PUBLIC.

These attributes also affect whether authentication is required during client sessions with ODBC, JDBC, or ESQL/C connections, in which the SET SESSION AUTHORIZATION statement attempts to switch to a different user ID after a trusted connection has been established.

DEFAULT ROLE attributes

A ROLE object specifies the user's role (and privileges) when using a trusted connection. A DEFAULT ROLE identifies a role that exists at the current server, and is used when a user does not have a user-specific role defined as part of the definition of the trusted-context object. The NO DEFAULT ROLE attribute will specify that the trusted-context object does not have a default role. The default is NO DEFAULT ROLE. The role explicitly specified for the user overrides any default role associated with the trusted-context object.

ENABLE and DISABLE keywords

The ENABLE keyword specifies that the trusted-context object is created in an enabled state.

The DISABLE keyword specifies that the new trusted-context object is created in a disabled state, and is not enabled for any new trusted connections that are established.

You cannot use the SET Database Object Mode statement of SQL to change the ENABLE or DISABLE attributes of trusted contexts. You must use the ALTER TRUSTED CONTEXT statement if you need to reset the ENABLED or DISABLED mode of a trusted-context.

Examples of trusted-context definitions

Example 1: Create a trusted-context object such that the current user on a trusted connection based on this trusted-context object can be switched to two different user IDs. When the current user of the connection is switched to `joe`, authentication is not required. However, authentication is required when the current user of the connection is switched to `bob`. Note that the trusted-context object has a default role called `MANAGER`. This implies that users working within the confines of this trusted-context object inherit the discretionary access privileges associated with the `MANAGER` role.

```
CREATE TRUSTED CONTEXT appserver
  USER wrjaibi
  DEFAULT ROLE MANAGER
  ENABLE
  ATTRIBUTES (ADDRESS '9.26.113.204')
  WITH USE FOR joe WITHOUT AUTHENTICATION,
    bob WITH AUTHENTICATION;
```

Example 2: Create a trusted-context object such that the current user of a trusted connection based on this trusted-context object can be switched to any user ID without authentication.

```
CREATE TRUSTED CONTEXT securerole
  USER pbird
  ENABLE
  ATTRIBUTES (ADDRESS 'example.ibm.com')
  WITH USE FOR PUBLIC WITHOUT AUTHENTICATION;
```

CREATE USER statement (UNIX™, Linux™)

Use the CREATE USER statement to define internally authenticated users, or to map externally authenticated users to surrogate user properties required for access to Informix® resources.

This statement is an extension to the ANSI/ISO standard for the SQL language.

Syntax

(explicit id unique_210_Connect_42_gract001) unique_210_Connect_42_gract001

```
CREATE { DEFAULT USER WITH [ { ACCOUNT LOCK | ACCOUNT UNLOCK } ] <Properties> | USERuser [ WITH [ PASSWORDpassword ] [ { ACCOUNT LOCK | ACCOUNT UNLOCK } ] <Properties> ] }
```

Properties

" PROPERTIES "

" { UIDuser_ID GROUP([ ] { | surrog_group_ID | surrog_group }) | USERsurrog_user [GROUP([ ] { | surrog_group_ID | surrog_group })] } "

" [HOME"directory "] "

" [AUTHORIZATION({ | DBSA | DBSSO | AAO | BARGROUP })] "

Element	Description	Restrictions	Syntax
directory	Path name of directory where user files are stored.	Must be 255 bytes or fewer, and must conform to the rules of your operating system. The *directory* must also: • Belong to the mapped *user_ID* and *surrog_group_ID*. • Have read, write, and execute permissions for the owner.	Quoted String on page 1202
password	Password for internal authentication of *user*.	Must be 6 - 32 bytes.	Quoted String on page 1202
surrog_group	Name of an existing operating system group (surrogate group) that has the permissions to which you want to map *user*. The list of *surrog_group* values must be enclosed in parentheses.	Must be 32 bytes or fewer. You must use one of the surrogates that are specified in the `/etc/informix/allowed.surrogates` file.	Owner name on page 1262
surrog_group_ID	Group identifier number (surrogate group) to which you want	The *surrog_group_ID* cannot be:	Literal Number on page 1198

Element	Description	Restrictions	Syntax
	to map the *user*. The list of *surrog_group_id* value or values that you specify must be enclosed in parentheses.	• A group ID with server administrative privileges (DBSA, DBSSO, AAO, and BARGROUP) • Group 0 (**root**, sometimes referred to as **wheel** or **system**) • Group 80 on Mac OS X (**admin**) • A group ID associated with group **bin** or group **sys** You must use one of the surrogates that are specified in the `/etc/informix/allowed.surrogates` file.	
surrog_user	Name of an existing OS user account (surrogate user) on the Informix® host computer that has the permissions to which you want to map *user*.	Must conform to the rules of your operating system. Must be one of the surrogates that are specified in file `/etc/informix/allowed.surrogates` file.	Owner name on page 1262
user	Authorization identifier of the specific user that you are mapping to user properties.	Cannot be PUBLIC.	Owner name on page 1262
user_ID	User identifier number to which to map *user*.	Cannot be that of user **root** or of user **informix**. Must be one of the surrogates that are specified in file `/etc/informix/allowed.surrogates`.	Literal Number on page 1198

Usage

Only a DBSA can run the CREATE USER statement. With a non-root installation, the user who installs the server is the equivalent of the DBSA, unless the user delegates DBSA privileges to a different user.

The USERMAPPING configuration parameter must be set to a value that enables support for mapped users before users defined by the CREATE USER statement can connect to the database server. A DBSA can issue the CREATE USER statement to map users to properties that correspond to the appropriate level of authorization.

You must also enter values in the SYSUSERMAP table of the **sysusers** database to map users with the appropriate user properties so that the mapped user statements of SQL to work correctly.

Execution of the CREATE USER statement can be audited with the CRUR audit code.

PASSWORD clause

For a root-privileged server, if an OS user is connecting and the USERMAPPING configuration parameter is unset, OS authentication occurs even though the user exists in the database. When the USERMAPPING parameter is set, internal user authentication takes precedence over OS authentication. Mapped users are authenticated internally or externally. When a user is created without a password, a mapped user is created. When a user is created with a password, an internally authenticated user is created with the properties from the operating system, unless an explicit PROPERTIES clause is also specified in the statement. When the CREATE USER statement contains both the PASSWORD clause and PROPERTIES clause, the user is an internally authenticated user, but has the surrogate properties that are specified in PROPERTIES clause. In this case, the surrogate user or group must also be listed in the `/etc/informix/allowed.surrogates file`.

PROPERTIES clause

The PROPERTIES clause can define a new user, and can optionally associate that user with surrogate properties that can include a group and a home directory. CREATE DEFAULT USER is a special case of the CREATE USER statement. The CREATE DEFAULT USER statement defines the properties that are set for the default user. After you define default user properties, you can create new users who have default user properties by omitting the PROPERTIES clause. Mapped users can connect to the database server with the surrogate user properties if they authenticate with pluggable authentication module (PAM), single sign-on (SSO), or internal authentication. Property values are not applicable to non-root installations but must be specified just like a root-privileged server. However, surrogate users and groups in non-root installations are not required in the `allowed.surrogates` file.

AUTHORIZATION clause

The AUTHORIZATION clause grants a subset of administrative privileges. The USERMAPPING configuration parameter must be set to ADMIN to enable this clause.

Note:

Use of this AUTHORIZATION clause (and of the AUTHORIZATION clause of the ALTER USER or GRANT ACCESS TO PROPERTIES statements) is not recommended. This syntax will not support role separation in a future release.

HOME directory clause

Specifying a directory for the user files with the HOME keyword is optional, but in some cases it is highly desirable. If you do not specify a home directory, an externally authenticated user has the same home directory as the surrogate user account on the Informix® host computer. If the surrogate user identity that does not have a set home directory, then Informix® creates a directory for user files in `$INFORMIXDIR/users`. In the latter case, the directory name in `$INFORMIXDIR/users` takes the form `uid.`*ID_number* (for example, `uid.101`).

ACCOUNT LOCK and ACCOUNT UNLOCK keywords

With the ACCOUNT LOCK and ACCOUNT UNLOCK keywords, the DBSA can toggle disabling and enabling the specified user's access to the database server.

Example

Examples

Example 1: Create a mapped user:

The following statement creates a mapped user named **joe**.

```
CREATE USER joe;
```

If the user **joe** is an OS user, **joe** has the operating system properties that are associated with his user name.

If the user **joe** is not an OS user and if default user properties are defined, **joe** has the surrogate properties of the default user. If default user properties are not defined, an error is returned.

Example 2: Create an internally authenticated user:

The following statement creates an internally authenticated user named **joe** with a password of **joebar**:

```
CREATE USER joe WITH PASSWORD "joebar";
```

If the user **joe** is not an OS user and if default user properties are defined, **joe** has the surrogate properties of the default user. If default user properties are not defined, an error is returned

Example 3: Create an internally authenticated user with a locked account:

The following statement creates an internally authenticated user named **phil** with a locked account:

```
CREATE USER phil WITH PASSWORD "joebar" ACCOUNT LOCK;
```

If the user **phil** is not an OS user and if default user properties are defined, **phil** has the surrogate properties of the default user. If default user properties are not defined, an error is returned

Example 4: Create an internally authenticated user with specific properties:

The following statement creates an internally authenticated user named **mary** with a UID, a group, and a home directory:

```
CREATE USER mary WITH PASSWORD "joebar" PROPERTIES UID 44567
GROUP(1234) HOME "/home/pd/osuser";
```

Example 5: Create a mapped user with a surrogate user:

The following statement creates a mapped user named **bill** with a surrogate user name of **foo_os**:

```
CREATE USER bill WITH PROPERTIES user "foo_os";
```

The user **bill** has the properties of the operating system user **foo_os**.

Example 6: Create a default user:

The following statement creates a user, internally named PUBLIC, with the properties of the surrogate user **tmp**:

```
CREATE DEFAULT USER WITH PROPERTIES USER "tmp";
```

Other users created without surrogate properties will have these properties.

CREATE VIEW statement

Use the CREATE VIEW statement to create a new view that is based on one or more existing tables and views that reside in the database, or in another database of the local database server or of a different database server.

Syntax

(explicit id unique_420_Connect_42_crv003) unique_420_Connect_42_crv003 **CREATE VIEW [IF NOT EXISTS]** *view [owner]* [{ (*column*) | **OF TYPE** *row_type* }] **AS** <Subset of SELECT Statement>[] [**WITH CHECK OPTION**]

Element	Description	Restrictions	Syntax
column	Name that you declare here for a column in view. Default is a column name from Projection list of SELECT.	See Naming View Columns on page 424.	Identifier on page 1231
owner	Owner of the view. If omitted, default is the user ID that issues the statement.	To specify another user ID requires DBA access privilege.	Owner name on page 1262
row_type	Named-row type for typed view	Must already exist in the database	Data Type on page 985
view	Name that you declare here for the view	Must be unique among view, table, sequence, and synonym names in the database.	Identifier on page 1231

Usage

A *view* is a virtual table, defined by a SELECT statement. Except for the statements in the following list, you can specify the name or synonym of a view in any SQL statement where the name of a table is syntactically valid:

- ALTER FRAGMENT
- CREATE INDEX
- CREATE TABLE
- CREATE TRIGGER
- RENAME TABLE
- START VIOLATIONS TABLE
- STOP VIOLATIONS TABLE
- TRUNCATE
- UPDATE STATISTICS

You must specify the name of a view when you use the CREATE TRIGGER statement to define an INSTEAD OF trigger on a view, but the syntax and functionality are different from those of a trigger defined on a table.

Updating Through Views on page 426 prohibits non-updatable views in INSERT, DELETE, or UPDATE statements (where other views are valid).

To create a view, you must have the Select privilege on all columns from which the view is derived. You can query a view as if it were a table, and in some cases, you can update it as if it were a table; but a view is not a table.

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if a view of the specified name is already registered in the current database, or if the specified name is the identifier of a table, synonym, or sequence object in the current database.

The view consists of the set of rows and columns that the SELECT statement in the view definition returns each time you refer to the view in a query.

In some cases, the database server merges the SELECT statement of the user with the SELECT statement defining the view and executes the combined statements. In other cases, a query against a view might execute more slowly than expected, if the complexity of the view definition causes the database server to create a temporary table (referred to as a materialized view). For more information on materialized views, see the .

The view reflects changes to the underlying tables, but with two exceptions:

- If a SELECT * specification defines the view, the view has only the columns that existed in the underlying tables when the view was defined by CREATE VIEW. Any new columns that are subsequently added to the underlying tables with the ALTER TABLE statement do not appear in the view.
- If a GRANT or REVOKE statement changes the discretionary access privileges on any table referenced in the view definition, the database server does not automatically apply those access privilege changes to the view.

To force modifications of the access privileges or schema of an underlying table to be applied to the view, you can use the DROP VIEW and CREATE VIEW statements of SQL to drop and recreate the view. You can also use the CREATE VIEW and CREATE TRIGGER statements to recreate, respectively, any dependent views or INSTEAD OF triggers that the DROP VIEW statement destroyed.

The view inherits the data types of the columns in the tables from which the view is derived. The database server determines data types of virtual columns from the nature of the expression.

The SELECT statement is stored in the **sysviews** system catalog table. When you subsequently refer to a view in another statement, the database server performs the defining SELECT statement while it executes the new statement.

In , if you create a view outside the CREATE SCHEMA statement, you receive warnings if you use the **-ansi** flag or if you set the **DBANSIWARN** environment variable.

The following statement creates a view that is based on the **person** table. When you create a view like this, which has no OF TYPE clause, the view is referred to as an *untyped view*.

```
CREATE VIEW v1 AS SELECT * FROM person;
```

Typed Views

You can create *typed views* if you have Usage privileges on the named-ROW type or if you are its owner or the DBA. If you omit the OF TYPE clause, rows in the view are considered untyped and default to an unnamed-ROW type.

Typed views, like typed tables, are based on a named-ROW type. Each column in the view corresponds to a field in the named-ROW type. The following statement creates a typed view that is based on the table **person**.

```
CREATE VIEW v2 OF TYPE person_t AS SELECT * FROM person;
```

To create a typed view, you must include an OF TYPE clause. When you create a typed view, the named-ROW type that you specify immediately after the OF TYPE keywords must already exist.

Subset of SELECT syntax valid in view definitions

Most SELECT statement syntax is supported in view definitions, with certain exceptions.

- You cannot create a view on a temporary table. The FROM clause of the SELECT clause of the view definition cannot include the name of a temporary table.
- Table objects referenced by the SELECT clause in CREATE VIEW statements can be permanent database tables, views, or derived tables. The query can reference a single table object, or can join two or more. These can be in the current database, in other databases of the local database server, or in databases of remote server instances. The SELECT statement can define derived tables in the FROM clause, using uncorrelated or correlated table references. These derived table definitions can include the `LATERAL` keyword, and can include lateral table and column references.
- Table objects referenced by the SELECT clause in CREATE VIEW statements can be permanent database tables, views, or derived tables. The query can reference a single table object, or can join two or more. These can be in the current database, in other databases of the local database server, or in databases of remote server instances. The SELECT statement can define derived tables in the FROM clause, using uncorrelated or correlated table references. These derived table definitions can include the `LATERAL` keyword, and can include lateral table and column references.

- If Select privileges are revoked from a user for a table that is referenced in the SELECT statement defining a view that the same user owns, then that view is automatically dropped by the database server, unless the view also includes columns from tables in another database.
- You cannot create a view on a table that is part of a typed-table hierarchy, if that table resides in a database of a different database server instance.
- Do not use display labels in the Select list of the Projection clause in a view definition. Display labels in the Projection clause are interpreted as column names.
- Hardcoded values should not be used in a view definition, but only in the WHERE clause of subsequent SELECT statements that query the view. If the values are not hardcoded in the view, the query optimizer can then always exclude those literal values and can complete the query in less time. But if the same values are hardcoded in the view, the query optimizer still must evaluate each literal value.
- The SELECT clause in the CREATE VIEW statement cannot include the SKIP, FIRST, or LIMIT keywords, or the INTO TEMP clause.

For complete information about SELECT statement syntax and usage, see SELECT statement on page 690.

Union Views

A view that contains a UNION or UNION ALL operator in its SELECT statement is known as a *union view*. Certain restrictions apply to union views:

- If a CREATE VIEW statement defines a union view, or includes the INTERSECT, MINUS, or EXCEPT set operator, you cannot specify the WITH CHECK OPTION keywords in the CREATE VIEW statement.
- All restrictions that apply to UNION or UNION ALL set operations in stand-alone SELECT statements also apply to UNION and UNION ALL operations in the SELECT statement of a union view.
- Similarly, all other restrictions that apply to the INTERSECT, MINUS, or EXCEPT set operators in stand-alone SELECT statements also apply to those set operators in a combined SELECT statement that defines a view.

For a list of these restrictions, see Restrictions on a Combined SELECT on page 756. For an example of a CREATE VIEW statement that defines a union view, see Naming View Columns on page 424.

Naming View Columns

The number of columns that you specify in the *column* list must match the number of columns returned by the SELECT statement that defines the view. If you do not specify a list of columns, the view inherits the column names of the underlying tables. In the following example, the view **herostock** has the same column names as the columns in Projection clause of the SELECT statement:

```
CREATE VIEW herostock AS
   SELECT stock_num, description, unit_price, unit, unit_descr
      FROM stock WHERE manu_code = 'HRO';
```

You must specify at least one column name in the following circumstances:

- If you provide names for some of the columns in a view, then you must provide names for all the columns. That is, the column list must contain an entry for every column that appears in the view.
- If the SELECT statement returns an expression, the corresponding column in the view is called a *virtual* column. You must provide a name for a virtual column. In the following example, the user must specify the column parameter because the select list of the Projection clause of the SELECT statement contains an aggregate expression:

```
CREATE VIEW newview (firstcol, secondcol) AS
         SELECT sum(cola), colb FROM oldtab;
```

- You must also specify column names in cases where any of the selected columns have duplicate column names without the table qualifiers. For example, if both **orders.order_num** and **items.order_num** appear in the SELECT statement, the CREATE VIEW statement, must provide two separate column names to label them:

```
CREATE VIEW someorders (custnum,ocustnum,newprice) AS
         SELECT orders.order_num,items.order_num,
               items.total_price*1.5
            FROM orders, items
            WHERE orders.order_num = items.order_num
            AND items.total_price > 100.00;
```

  Here **custnum** and **ocustnum** replace the two identical column names.

- The CREATE VIEW statement must also provide column names in the column list when the SELECT statement includes a UNION or UNION ALL operator and the names of the corresponding columns in the SELECT statements are not identical.

  For example, code in the following CREATE VIEW statement must specify the column list because the second column in the first SELECT statement has a different name from the second column in the second SELECT statement:

```
CREATE VIEW myview (cola, colb) AS
         SELECT colx, coly from firsttab
         UNION
         SELECT colx, colz from secondtab;
```

Using a View in the SELECT Statement

You can define a view whose columns are based on other views, but you must abide by the restrictions on creating views that are discussed in the *IBM® Informix® Database Design and Implementation Guide*.

WITH CHECK OPTION Keywords

The WITH CHECK OPTION keywords instruct the database server to ensure that all modifications that are made through the view to the underlying tables satisfy the definition of the view.

The following example creates a view that is named **palo_alto**, which uses all the information in the **customer** table for customers in the city of Palo Alto. The database server checks any modifications made to the **customer** table through **palo_alto** because the WITH CHECK OPTION keywords are specified.

```
CREATE VIEW palo_alto AS
   SELECT * FROM customer WHERE city = 'Palo Alto'
      WITH CHECK OPTION
```

You can insert into a view a row that does not satisfy the conditions of the view (that is, a row that is not visible through the view). You can also update a row of a view so that it no longer satisfies the conditions of the view. For example, if the view was created without the WITH CHECK OPTION keywords, you could insert a row through the view where the city is Los Altos, or you could update a row through the view by changing the city from Palo Alto to Los Altos.

To prevent such inserts and updates, you can add the WITH CHECK OPTION keywords when you create the view. These keywords ask the database server to test every inserted or updated row to ensure that it meets the conditions that are set by the WHERE clause of the view. The database server rejects the operation with an error if the row does not meet the conditions.

Even if the view was created with the WITH CHECK OPTION keywords, however, you can perform inserts and updates through the view to change columns that are not part of the view definition. A column is not part of the view definition if it does not appear in the WHERE clause of the SELECT statement that defines the view.

The CREATE VIEW statement fails with error -940 if you include the WITH CHECK OPTION keywords in a CREATE VIEW statement in which the UNION, INTERSECT, MINUS, or EXCEPT set operator combines two queries in the view definition.

Updating Through Views

If a view is built on a single table, the view is *updatable* if the SELECT statement that defines the view does not contain any of the following elements:

- Columns in the projection list that are aggregate values
- Columns in the projection list that use the UNIQUE or DISTINCT keyword
- A GROUP BY clause
- A UNION operator
- A query that selects calculated or literal values.

You can DELETE from a view that selects calculated values from a single table, but INSERT and UPDATE operations are not valid on such views.

In an updatable view, you can update the values in the underlying table by inserting values into the view. If a view is built on a table that has a derived value for a column, however, that column is not updatable through the view. Other columns in the view, however, can be updated.

See also Updating Views for information about using INSTEAD OF triggers to update views that are based on more than one table or that include columns containing aggregates or other calculated values.

Important: You cannot update or insert rows in a remote table through views that were created using the WITH CHECK OPTION keywords.

CREATE XADATASOURCE statement

Use the CREATE XADATASOURCE statement to create a new XA-compliant data source and create an entry for it in the **sysxadatasources** system catalog table. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
CREATE XADATASOURCE [IF NOT EXISTS] xa_source USINGxa_type
```

Element	Description	Restrictions	Syntax
xa_source	Name that you declare here for the new XA data source	Must be unique among XA data source names in **sysxadatasources**	Identifier on page 1231
xa_type	Name of an existing XA data source type	Must already exist in the database in the **sysxasourcetypes** system catalog table	Identifier on page 1231

Usage

An XA-compliant data source is an external data source that complies with the X/Open DTP XA Standard for managing interactions between a transaction manager and a resource manager. To register XA-compliant data sources in the database requires two SQL statements:

- First create one or more XA-compliant data source types by using the CREATE XADATASOURCE TYPE statement.
- Then create one or more instances of XA-compliant data sources with the CREATE XADATASOURCE statement.

You can integrate transactions at the XA data source with the Informix® transaction, using a 2-phase commit protocol to ensure that transactions are uniformly committed or rolled back among multiple database servers, and manage multiple external XA data sources within the same global transaction.

The CREATE XADATASOURCE statement is not supported on secondary servers within a high-availability cluster.

Any user can create an XA data source, which follows standard owner-naming rules, according to the ANSI-compliance status of the database. Only a database that uses transaction logging can support the X/Open DTP XA Standard.

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if an XA data source of the specified name is already registered in the current database.

Both XA data source types and instances of XA data sources are specific to one database. To support distributed transactions, they must be created in each database that interacts with the external XA data source.

The following statement creates a new XA data source instance called **informix.NewYork** of type **informix.MQSeries**.

```
CREATE XADATASOURCE informix.NewYork USING informix.MQSeries;
```

SQL statements that are not valid in XA environments

The Informix® database server issues error -701 when you attempt to execute any of the following statements within an X/Open distributed transaction-processing environment:

- CLOSE DATABASE
- CREATE DATABASE
- DROP DATABASE
- SET LOG

- SAVEPOINT
- RELEASE SAVEPOINT
- ROLLBACK TO SAVEPOINT

Within an XA environment, you can execute a single DATABASE statement after an xa_open call to specify a current database. After this database is selected, however, no subsequent DATABASE statement can be executed in the same session. If you attempt to execute a second DATABASE statement, the DATABASE statement fails with error -701.

CREATE XADATASOURCE TYPE statement

Use the CREATE XADATASOURCE TYPE statement to create a new XA-compliant data source type and create an entry for it in the **sysxasourcetypes** system catalog table. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_586_Connect_42_xat002) unique_586_Connect_42_xat002 **CREATE XADATASOURCE TYPE [IF NOT EXISTS]** *xa_type* (<Purpose Options>[])

Element	Description	Restrictions	Syntax
xa_type	Name that you declare here for a new XA data source type	Must be unique among XA data source type names in the **sysxasourcetypes** system catalog table	Identifier on page 1231

Usage

The CREATE XADATASOURCE TYPE statement adds an XA-compliant data source type to the database.

The CREATE XADATASOURCE TYPE statement is not supported on secondary servers within a high-availability cluster.

Any user can create an XA data source type, whose owner-naming rules depend on the ANSI-compliance status of the database. Only a database that uses transaction logging can support the X/Open DTP XA Standard.

To create a data source type, you must declare its name and specify *purpose functions* and *purpose values* as attributes of the XA source type. Most of the purpose options that follow the source type name associate columns in the **sysxasourcetypes** system catalog table with the name of a UDR.

If you include the optional IF NOT EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if an XA data source type of the specified name is already registered in the current database.

Both XA data source types and instances of XA data sources are specific to one database. To support distributed transactions, they must be created in each database that interacts with the external XA data source.

The following statement creates a new XA data source type called **MQSeries®**, owned by user **informix**.

```
CREATE XADATASOURCE TYPE 'informix'.MQSeries(
                          xa_flags    = 1,
                          xa_version  = 0,
                          xa_open     = informix.mqseries_open,
                          xa_close    = informix.mqseries_close,
```

```
                          xa_start    = informix.mqseries_start,
                          xa_end      = informix.mqseries_end,
                          xa_rollback = informix.mqseries_rollback,
                          xa_prepare  = informix.mqseries_prepare,
                          xa_commit   = informix.mqseries_commit,
                          xa_recover  = informix.mqseries_recover,
                          xa_forget   = informix.mqseries_forget,
                         xa_complete = informix.mqseries_complete);
```

You need to provide one value or UDR name for each of the options listed above, but the sequence in which you list them is not critical. (The order of purpose options in this example corresponds to the order of column names in the **sysxasourcetypes** system catalog table.)

After this statement executes successfully, you can create instances of type **informix.MQSeries**. The following statement creates a new instance called **informix.MenloPark** of XA-compliant data source type **informix.MQSeries:**

```
CREATE XADATASOURCE informix.MenloPark USING informix.MQSeries;
```

DATABASE statement

Use the DATABASE statement to open an accessible database as the current database. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

DATABASE*database* [**EXCLUSIVE**]

Element	Description	Restrictions	Syntax
database	Name of a database to which to connect	The database must exist	Database Name on page 1224

Usage

You can use the DATABASE statement to select any database of your database server instance. To select a database on another database server, include the name of the database server with the database name:

```
DATABASE stores_demo@db_titinius;
```

You can also specify the database as a host variable that contains a valid database environment:

```
DATABASE db_varX;
```

Here the contents of **db_varX** must correspond to one of the formats in the Database Name on page 1224 syntax diagram.

If besides the name of the *database* you also provide the name of the current (or of another) database server instance, the database server name cannot include uppercase characters.

If the DATABASE statement specifies only a *database* name that corresponds to no database of the current server instance, the database server examines the **DBPATH** environment variable setting, and opens the specified database if a server is found. If no *database* is found, or if its server is offline, the DATABASE statement fails with an error. For more information, see the CONNECT statement topic Only Database Specified on page 179.

Issuing the DATABASE statement when a database is already open closes the current database before opening the new one. Closing the current database releases any cursor resources that the database server holds, invalidating any cursors that you have declared up to that point. If the *user* specification was changed through a SET SESSION AUTHORIZATION statement, the original user name is restored when the new database is opened.

If a previous CONNECT statement has established an explicit connection to a database, and that connection is still your current connection, you cannot use the DATABASE statement (nor any SQL statement that creates an implicit connection) until after you use DISCONNECT to close the explicit connection.

The current user (or PUBLIC) must have the Connect privilege on the database that is specified in the DATABASE statement. The current user cannot have the same user name as an existing role in the database.

DATABASE is not a valid statement in multistatement PREPARE operations.

You can use the 'dbhostname' option to retrieve the host name of the database server to which a database client is connected. This option retrieves the physical computer name of the computer on which the database server is running. In the following example, the user enters the 'dbhostname' option of DBINFO in a SELECT statement to retrieve the host name of the database server instance to which **DB-Access** is connected:

```
SELECT DBINFO('dbhostname')
   FROM systables
   WHERE tabid = 1;
```

The following table shows the result of this query.

(constant)
rd_lab1

SQLCA.SQLWARN Settings Immediately after DATABASE Executes (ESQL/C)

Immediately after DATABASE executes, you can identify characteristics of the specified database by examining warning flags in the **sqlca** structure.

- If the first field of **sqlca.sqlwarn** is blank, then no warnings were issued.
- The second **sqlca.sqlwarn** field is set to the letter `w` if the *database* that was opened supports transaction logging.
- The third field is set to `w` if *database* is an ANSI-compliant database.
- The fourth field is set to `w`.
- The fifth field is set to `w` if *database* converts all floating-point data to DECIMAL format. (System lacks FLOAT and SMALLFLOAT support.)
- The seventh field is set to `w` if *database* is the secondary server (that is, running in read-only mode) in a data-replication pair.
- The eighth field is set to `w` if *database* has **DB_LOCALE** set to a locale different from the **DB_LOCALE** setting on the client system.

EXCLUSIVE keyword

The EXCLUSIVE keyword opens the database in exclusive mode and prevents access by anyone but the current user. To allow others to access the database, you must first execute the CLOSE DATABASE statement and then reopen the database without the EXCLUSIVE keyword.

The following statement opens the **stores_demo** database on the **training** database server in exclusive mode:

```
DATABASE stores_demo@training EXCLUSIVE;
```

If another user has already opened the specified database, exclusive access is denied, an error is returned, and no database is opened.

If you encounter this error, but you are unable to confirm that other users are connected to the database, your non-exclusive access might be caused by a sensor or task that is running in the Scheduler API. To temporarily disable the Scheduler, you can issue this SQL administration API command:

```
EXECUTE FUNCTION admin('scheduler shutdown');
```

After the admin('scheduler shutdown') routine has completed execution. retry the `DATABASE ... EXCLUSIVE` statement.

For more information on the Scheduler API commands, see your *HCL® Informix® Administrator's Guide*. For information about the privileges that you must hold to call SQL administration API functions, see your *HCL® Informix® Administrator's Reference*.

DEALLOCATE COLLECTION statement

Use the DEALLOCATE COLLECTION statement to release memory for a collection variable that was previously allocated with the ALLOCATE COLLECTION statement.

This statement is an extension to the ANSI/ISO standard for SQL. Use this statement with ESQL/C.

Syntax

DEALLOCATE COLLECTION :*variable*

Element	Description	Restrictions	Syntax
variable	Name that identifies a typed or untyped collection variable for which to deallocate memory	Must be the name of the collection variable that has already been allocated	Name must conform to language-specific rules for names of variables

Usage

The DEALLOCATE COLLECTION statement frees all the memory that is associated with the collection variable that *variable* identifies. You must explicitly release memory resources for a collection variable with DEALLOCATE COLLECTION. Otherwise, deallocation occurs automatically at the end of the program.

The DEALLOCATE COLLECTION statement releases resources for both typed and untyped collection variables.

Tip: The DEALLOCATE COLLECTION statement deallocates memory for collection variables only. To deallocate memory for row variables, use the DEALLOCATE ROW statement.

If you deallocate nonexistent collection variables or variables that are not collection variables, you receive errors. After you deallocate a collection variable, you can use the ALLOCATE COLLECTION to reallocate resources and you can then reuse a collection variable.

This example shows how to deallocate resources with the DEALLOCATE COLLECTION statement for the untyped collection variable, **a_set**:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection a_set;
EXEC SQL END DECLARE SECTION;
. . .
EXEC SQL allocate collection :a_set;
. . .
EXEC SQL deallocate collection :a_set;
```

For a related example, see the related concept, Inserting into a Collection Cursor.

DEALLOCATE DESCRIPTOR statement

Use the DEALLOCATE DESCRIPTOR statement to free a previously allocated, system-descriptor area. Use this statement with .

Syntax

This statement is an extension to the ANSI/ISO standard for the SQL language.

```
DEALLOCATE DESCRIPTOR { 'descriptor' | descriptor_var }
```

Element	Description	Restrictions	Syntax
descriptor	Name of a system-descriptor area	Use single quotation marks. System-descriptor area must already be allocated	Quoted String on page 1202
descriptor_var	Host variable that contains the name of a system-descriptor area	System-descriptor area must already be allocated, and the variable must already have been declared	Name must conform to language-specific rules

Usage

The DEALLOCATE DESCRIPTOR statement frees all the memory that is associated with the system-descriptor area that *descriptor* or *descriptor_var* identifies. It also frees all the item descriptors (including memory for data items in the item descriptors).

You can reuse a descriptor or descriptor variable after it is deallocated. Otherwise, deallocation occurs automatically at the end of the program.

If you deallocate a nonexistent descriptor or descriptor variable, an error results.

You cannot use the DEALLOCATE DESCRIPTOR statement to deallocate an **sqlda** structure. You can use it only to free the memory that is allocated for a system-descriptor area.

The following examples show valid DEALLOCATE DESCRIPTOR statements. The first line uses an embedded-variable name, and the second line uses a quoted string to identify the allocated system-descriptor area.

```
EXEC SQL deallocate descriptor :descname;

EXEC SQL deallocate descriptor 'desc1';
```

DEALLOCATE ROW statement

Use the DEALLOCATE ROW statement to release memory for a ROW variable.

This statement is an extension to the ANSI/ISO standard for SQL. Use this statement with ESQL/C.

Syntax

DEALLOCATE ROW :*variable*

Element	Description	Restrictions	Syntax
variable	Typed or untyped row variable	Must be declared and allocated	Language specific

Usage

DEALLOCATE ROW frees all the memory that is associated with the typed or untyped **row** variable that *variable* identifies. If you do not explicitly release memory resources with DEALLOCATE ROW, deallocation occurs automatically at the end of the program. To deallocate memory for the collection variable, use the DEALLOCATE COLLECTION statement.

After you deallocate a ROW variable, you can use the ALLOCATE ROW statement to reallocate resources, and you can then reuse a ROW variable. The following example shows how to deallocate resources for the ROW variable, **a_row**, using the DEALLOCATE ROW statement:

```
EXEC SQL BEGIN DECLARE SECTION; row (a int, b int) a_row;
EXEC SQL END DECLARE SECTION;
. . .
EXEC SQL allocate row :a_row;
. . .
EXEC SQL deallocate row :a_row;
```

DECLARE statement

Use the DECLARE statement of dynamic SQL to declare a cursor and to associate it with an SQL statement that returns a set of rows to an SPL or or routine.

Syntax

(explicit id unique_223_Connect_42_decl002) unique_223_Connect_42_decl002(explicit id unique_223_Connect_42_decl003)
unique_223_Connect_42_decl003(explicit id unique_223_Connect_42_decl005) unique_223_Connect_42_decl005(explicit id unique_223_Connect_42_decl006)
unique_223_Connect_42_decl006(explicit id unique_223_Connect_42_decl007) unique_223_Connect_42_decl007(explicit id unique_223_Connect_42_decl008)
unique_223_Connect_42_decl008(explicit id unique_223_Connect_42_decl009) unique_223_Connect_42_decl009(explicit id unique_223_Connect_42_decl010)
unique_223_Connect_42_decl010(explicit id unique_223_Connect_42_decl011) unique_223_Connect_42_decl011 **DECLARE** { *cursor_id* |
[] *cursor_id_var* } { **CURSOR** [[] **WITH HOLD**] { **FOR** [] <Subset of INSERT Statement> [] <FOR UPDATE or FOR READ ONLY Select
Options> <Other Select or Function Options> } | [] [] **SCROLL CURSOR** [**WITH HOLD**] <Other Select or Function Options> | [] **CURSOR**
FOR { <SELECT with Collection-Derived Table> [] | <INSERT with Collection-Derived Table> [] } }

FOR UPDATE or FOR READ ONLY Select Options

" [] <Subset of SELECT Statement> [] { **FOR READ ONLY** | [] **FOR UPDATE** [**OF** *column*] } "

Other Select or Function Options

" **FOR** { [] <SELECT Statement> [] | *statement_id* | [] { [] | *statement_id_var* | <EXECUTE PROCEDURE Statement> [] } | []
<EXECUTE FUNCTION Statement> [] } "

Element	Description	Restrictions	Syntax
column	Column to update with cursor	Must exist, but need not be listed in Select list of the Projection clause	Identifier on page 1231
cursor_id	Name declared here for cursor	Must be unique in the routine among names of cursors and prepared objects (and in SPL, among variables)	Identifier on page 1231
cursor_id_var	Variable holding *cursor_id*	Must have a character data type	Language-specific
statement_id	Name of prepared statement	Must have been declared by a PREPARE statement	Identifier on page 1231
statement_id_var	Variable holding *statement_id*	Must have a character data type	Language-specific

Usage

Except as noted, sections that follow describe how to use the DECLARE statement in routines. For information about the more restricted syntax and semantics of the DECLARE statement in SPL routines, see Declaring a Dynamic Cursor in an SPL Routine on page 449.

A *cursor* is an identifier that you associate with an SQL statement that returns a group of rows. The DECLARE statement associates the cursor with one of the following database objects:

- With an SQL statement, such as SELECT, EXECUTE FUNCTION (or EXECUTE PROCEDURE), or INSERT.

  Each of these SQL statements creates a different type of cursor. For more information, see Overview of Cursor Types on page 436.

- With the statement identifier (*statement id* or *statement id variable*) of a prepared statement

  You can prepare one of the previous SQL statements and associate the prepared statement with a cursor. For more information, see Associating a Cursor with a Prepared Statement on page 445.

- With a collection variable in programs

  The name of the collection variable appears in the FROM clause of a SELECT or the INTO clause of an INSERT. For more information, see Associating a Cursor with a Prepared Statement on page 445.

DECLARE assigns an identifier to the cursor, specifies its uses, and directs the preprocessor to allocate storage for it. DECLARE must precede any other statement that references the cursor during program execution.

The cursors and prepared objects that can exist concurrently in a single program are limited by available memory. To avoid exceeding the limit, use the FREE statement to release the resources of prepared statements or cursors that are no longer needed.

An ESQL/C program can consist of one or more source-code files. By default, the scope of reference of a cursor is global to a program, so a cursor that was declared in one source file can be referenced from a statement in another file. If you want to limit the scope of each cursor name to the file where it was declared, you must preprocess each file with the -local command-line option.

Multiple cursors can be declared for the same prepared statement identifier. For example, the following example does not return an error:

```
EXEC SQL prepare id1 from 'select * from customer';
EXEC SQL declare x cursor for id1;
EXEC SQL declare y scroll cursor for id1;
EXEC SQL declare z cursor with hold for id1;
```

If you include the -**ansi** compilation flag (or if **DBANSIWARN** is set), warnings are generated for statements that use dynamic cursor names, dynamic statement identifiers, or statements that use collection-derived tables. Some error checking is performed at runtime, such as these typical checks:

- Invalid use of sequential cursors as scroll cursors
- Use of undeclared cursors
- Invalid cursor names or statement names (empty)

Checks for multiple declarations of a cursor of the same name are performed at compile time only if the cursor or statement is specified as an identifier. The following example uses a host variable to store the cursor name:

```
EXEC SQL declare x cursor for select * from customer;
. . .
stcopy("x", s);
EXEC SQL declare :s cursor for select * from customer;
```

A cursor uses the collating order that was in effect when the cursor was declared, even if this is different from the collation of the session at run time.

Overview of Cursor Types

Cursors are typically required for data manipulation language (DML) operations on more than one row of data (or on collection variables). You can declare the following types of cursors with the DECLARE statement:

- A *Select cursor* is a cursor associated with a SELECT statement.
- A *Function cursor* is a cursor associated with an EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement.
- An *Insert cursor* is a cursor associated with an INSERT statement.

Sections that follow describe each of these cursor types. Cursors can also have *sequential*, *scroll*, and *hold* characteristics (but an Insert cursor cannot be a scroll cursor). These characteristics determine the structure of the cursor; see Cursor Characteristics on page 441. In addition, a Select or Function cursor can specify *read-only* or *update* mode. For more information, see Select Cursor or Function Cursor on page 436.



Tip: Function cursors behave the same as Select cursors that are enabled as update cursors.

Besides associating a cursor directly with the text of an SQL statement, the FOR keyword of the DECLARE statement can be followed by the identifier of a prepared SQL statement, associating the cursor with the result set of an INSERT, SELECT, EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement that was prepared dynamically. This feature enables you to associate different SQL statements with the same cursor at different times. In this case, the type of cursor depends on the prepared SQL statement that the *statement_id* or *statement_id* variable specifies when the cursor is opened. (For more information, see Associating a Cursor with a Prepared Statement on page 445.)

Select Cursor or Function Cursor

When an SQL statement returns more than one group of values to the program, you must declare a cursor to save the multiple groups, or rows, of data and to access these rows one at a time. You must associate one of the following SQL statements with a cursor:

- If you associate a SELECT statement with a cursor, the cursor is called a *Select cursor*.

  A Select cursor is a data structure that represents a specific location within the active set of rows that the SELECT statement retrieved.

- If you associate an EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement with a cursor, the cursor is called a *Function cursor*.

  The Function cursor represents the columns or values that a user-defined function returns. Function cursors behave the same as Select cursors that are enabled as update cursors.

In Informix®, for compatibility with legacy applications, if an SPL function was created with the CREATE PROCEDURE statement, you can create a Function cursor with the EXECUTE PROCEDURE statement. With external functions, you must use the EXECUTE FUNCTION statement.

When you associate a SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement with a cursor, the statement can include an INTO clause. However, if you prepare the SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement, you must omit the INTO clause in the PREPARE statement and use the INTO clause of the FETCH statement to retrieve the values from the Collection cursor.

A Select or Function cursor can scan returned rows of data and to move data row by row into a set of receiving variables, as the following steps describe:

1. DECLARE

   Use DECLARE to define a cursor and associate it with a statement.

2. OPEN

   Use OPEN to open the cursor. The database server processes the query until it locates or constructs the first row of the active set.

3. FETCH

   Use FETCH to retrieve successive rows of data from the cursor.

4. CLOSE

   Use CLOSE to close the cursor when its active set is no longer needed.

5. FREE

   Use FREE to release the resources that are allocated for the cursor.

Using the FOR READ ONLY Option

Use the FOR READ ONLY keywords to define a cursor as a read-only cursor. A cursor declared to be read-only cannot be used to update (or delete) any row that it fetches.

The need for the FOR READ ONLY keywords depends on whether your database is ANSI compliant or not ANSI compliant.

In a database that is not ANSI compliant, the cursor that the DECLARE statement defines is a read-only cursor by default, so you do not need to specify the FOR READ ONLY keywords if you want the cursor to be a read-only cursor. The only advantage of specifying the FOR READ ONLY keywords explicitly is for better program documentation.

In an ANSI-compliant database, the cursor associated with a SELECT statement through the DECLARE statement is an update cursor by default, provided that the SELECT statement conforms to all of the restrictions for update cursors listed in Subset of SELECT statements associated with cursors on page 444. If you want a Select cursor to be read-only, you must use the FOR READ ONLY keywords when you declare the cursor.

The database server can use less stringent locking for a read-only cursor than for an update cursor.

The following example creates a read-only cursor:

```
EXEC SQL declare z_curs cursor for
   select * from customer_ansi
   for read only;
```

Using the FOR UPDATE Option

Use the FOR UPDATE option to declare an update cursor. You can use the update cursor to modify (update or delete) the current row.

In an ANSI-compliant database, you can use a Select cursor to update or delete data if the cursor was not declared with the FOR READ ONLY keywords and it follows the restrictions on update cursors that are described in Subset of SELECT statements associated with cursors on page 444. You do not need to use the FOR UPDATE keywords when you declare the cursor.

The following example declares an update cursor:

```
EXEC SQL declare new_curs cursor for
   select * from customer_notansi
   for update;
```

In an update cursor, you can update or delete rows in the active set. After you create an update cursor, you can update or delete the currently selected row by using an UPDATE or DELETE statement with the WHERE CURRENT OF clause. The words CURRENT OF refer to the row that was most recently fetched; they take the place of the usual test expressions in the WHERE clause.

An update cursor lets you perform updates that are not possible with the UPDATE statement because the decision to update and the values of the new data items can be based on the original contents of the row. Your program can evaluate or manipulate the selected data before it decides whether to update. The UPDATE statement cannot interrogate the table that is being updated.

You can specify particular columns that can be updated. The columns need not appear in the Select list of the Projection clause.

Using FOR UPDATE with a List of Columns

When you declare an update cursor, you can limit the update to specific columns by including the OF keyword and a list of columns. You can modify only those named columns in subsequent UPDATE statements. The columns need not be in the select list of the SELECT clause.

The next example declares an update cursor and specifies that this cursor can update only the **fname** and **lname** columns in the **customer_notansi** table:

```
EXEC SQL declare name_curs cursor for
   select * from customer_notansi
   for update of fname, lname;
```

By default, unless declared as FOR READ ONLY, a Select cursor in a database that is ANSI compliant is an update cursor, so the FOR UPDATE keywords are optional. If you want an update cursor to be able to modify only some of the columns in a table, however, you must specify these columns in the FOR UPDATE OF *column* list.

The principal advantage to specifying columns is documentation and preventing programming errors. (The database server refuses to update any other columns.) An additional advantage is improved performance, when the SELECT statement meets the following criteria:

- The SELECT statement can be processed using an index.
- The columns that are listed are not part of the index that is used to process the SELECT statement.

If the columns that you intend to update are part of the index that is used to process the SELECT statement, the database server keeps a list of each updated row, to ensure that no row is updated twice. If the OF keyword specifies which columns can be updated, the database server determines whether or not to keep the list of updated rows. If the database server determines that the work of keeping the list is unnecessary, performance improves. If you do not use the OF *column* list, the database server always maintains a list of updated rows, although the list might be unnecessary.

The following example contains code that uses an update cursor with a DELETE statement to delete the current row.

Whenever the row is deleted, the cursor remains between rows. After you delete data, you must use a FETCH statement to advance the cursor to the next row before you can refer to the cursor in a DELETE or UPDATE statement.

```
EXEC SQL declare q_curs cursor for
   select * from customer where lname matches :last_name for update;

EXEC SQL open q_curs;
for (;;)
{
   EXEC SQL fetch q_curs into :cust_rec;
   if (strncmp(SQLSTATE, "00", 2) != 0)
      break;

   /* Display customer values and prompt for answer */
   printf("\n%s %s", cust_rec.fname, cust_rec.lname);
   printf("\nDelete this customer? ");
   scanf("%s", answer);

   if (answer[0] == 'y')
      EXEC SQL delete from customer where current of q_curs;
   if (strncmp(SQLSTATE, "00", 2) != 0)
      break;
}
printf("\n");
EXEC SQL close q_curs;
```

Locking with an Update Cursor

The FOR UPDATE keywords notify the database server that updating is possible and cause it to use more stringent locking than with a Select cursor. You declare an update cursor to let the database server know that the program might update (or delete) any row that it fetches as part of the SELECT statement. The update cursor employs *promotable* locks for rows that the program fetches. Other programs can read the locked row, but no other program can place a promotable lock (also called a *write lock*). Before the program modifies the row, the row lock is promoted to an exclusive lock.

It is possible to declare an update cursor with the WITH HOLD keywords, but the only reason to do so is to break a long series of updates into smaller transactions. You must fetch and update a particular row in the same transaction.

If an operation involves fetching and updating a large number of rows, the lock table that the database server maintains can overflow. The usual way to prevent this overflow is to lock the entire table that is being updated. If this action is impossible, an alternative is to update through a hold cursor and to execute COMMIT WORK at frequent intervals. You must plan such an application carefully, however, because COMMIT WORK releases all locks, even those that are placed through a hold cursor.

Subset of INSERT Statement with a Sequential Cursor

As indicated in the diagram for DECLARE statement on page 433, to create an Insert cursor, you associate a sequential cursor with a restricted form of the INSERT statement. The INSERT statement must include a VALUES clause; it cannot contain an embedded SELECT statement.

The following example contains code that declares an Insert cursor:

```
EXEC SQL declare ins_cur cursor for
   insert into stock values
   (:stock_no,:manu_code,:descr,:u_price,:unit,:u_desc);
```

The Insert cursor simply inserts rows of data; it cannot be used to fetch data. When an Insert cursor is opened, a buffer is created in memory. The buffer receives rows of data as the program executes PUT statements. The rows are written to disk only when the buffer is full. You can flush the buffer (that is, to write its contents into the database) when it is less than full, using the CLOSE, FLUSH, or COMMIT WORK statements. This topic is discussed further under the CLOSE, FLUSH, and PUT statements.

You must close an Insert cursor to insert any buffered rows into the database before the program ends. You can lose data if you do not close the cursor properly. For a complete description of INSERT syntax and usage, see INSERT statement on page 580.

Insert Cursor

About this task

When you associate an INSERT statement with a cursor, the cursor is called an *Insert cursor*. An Insert cursor is a data structure that represents the rows that the INSERT statement is to add to the database. The Insert cursor simply inserts rows of data; it cannot be used to fetch data. To create an Insert cursor, you associate a cursor with a restricted form of the INSERT statement. The INSERT statement must include a VALUES clause; it cannot contain an embedded SELECT statement.

Create an Insert cursor if you want to add multiple rows to the database in an INSERT operation. An Insert cursor allows bulk insert data to be buffered in memory and written to disk when the buffer is full, as these steps describe:

1. Use DECLARE to define an Insert cursor for the INSERT statement.
2. Open the cursor with the OPEN statement. The database server creates the insert buffer in memory and positions the cursor at the first row of the insert buffer.
3. Copy successive rows of data into the insert buffer with the PUT statement.
4. The database server writes the rows to disk only when the buffer is full. You can use the CLOSE, FLUSH, or COMMIT WORK statement to flush the buffer when it is less than full. This topic is discussed further under the PUT and CLOSE statements.
5. Close the cursor with the CLOSE statement when the insert cursor is no longer needed. You must close an Insert cursor to insert any buffered rows into the database before the program ends. You can lose data if you do not close the cursor properly.
6. Free the cursor with the FREE statement. The FREE statement releases the resources that are allocated for an Insert cursor.

Using an Insert cursor is more efficient than embedding the INSERT statement directly. This process reduces communication between the program and the database server and also increases the speed of the insertions.

Insert cursors also have the sequential cursor characteristic. To create an Insert cursor, you associate a sequential cursor with a restricted form of the INSERT statement. (For more information, see Insert Cursor on page 440.) The following example contains code that declares a sequential Insert cursor:

```
EXEC SQL declare ins_cur cursor for
   insert into stock values
   (:stock_no,:manu_code,:descr,:u_price,:unit,:u_desc);
```

Cursor Characteristics

You can declare a cursor as a *sequential* cursor (the default), a *scroll* cursor (by using the SCROLL keyword), or a *hold* cursor (by using the WITH HOLD keywords). The SCROLL and WITH HOLD keywords are not mutually exclusive. Sections that follow explain these structural characteristics.

A Select or Function cursor can be either a sequential or a scroll cursor. An Insert cursor can only be a sequential cursor. In ESQL/C routines, Select, Function, and Insert cursors can optionally be hold cursors. (In SPL routines, all cursors are sequential cursors, but only Select cursors can be hold cursors.)

Creating a Sequential Cursor by Default

If you use only the CURSOR keyword, you create a sequential cursor, which can fetch only the next row in sequence from the active set. The sequential cursor can read through the active set only once each time it is opened.

If you are using a sequential cursor for a Select cursor, on each execution of the FETCH statement, the database server returns the contents of the current row and locates the next row in the active set.

The following example creates a read-only sequential cursor in a database that is not ANSI compliant and an update sequential cursor in an ANSI-compliant database:

```
EXEC SQL declare s_cur cursor for
   select fname, lname into :st_fname, :st_lname
   from orders where customer_num = 114;
```

Insert cursors also have the sequential cursor characteristic. To create a Insert cursor, you associate a sequential cursor with a restricted form of the INSERT statement. (For more information, see Insert Cursor on page 440.) The following example declares an Insert cursor:

```
EXEC SQL declare ins_cur cursor for
   insert into stock values
   (:stock_no,:manu_code,:descr,:u_price,:unit,:u_desc);
```

Using the SCROLL Keyword to Create a Scroll Cursor

Use the SCROLL keyword to create a scroll cursor, which can fetch rows of the active set in any sequence.

The database server retains the active set of the cursor as a temporary table until the cursor is closed. You can fetch the first, last, or any intermediate rows of the active set as well as fetch rows repeatedly without having to close and reopen the cursor. (See FETCH.)

On a multiuser system, the rows in the tables from which the active-set rows were derived might change after the cursor is opened and a copy is made in the temporary table. If you use a scroll cursor within a transaction, you can prevent copied rows from changing either by setting the isolation level to Repeatable Read or by locking the entire table in share mode during the transaction. (See SET ISOLATION and LOCK TABLE.)

The following example creates a scroll cursor for a SELECT statement:

```
DECLARE sc_cur SCROLL CURSOR FOR SELECT * FROM orders;
```

You can create scroll cursors as Select and Function cursors but *not* as Insert cursors. Scroll cursors cannot be declared as FOR UPDATE.

Using the WITH HOLD keywords to create a hold cursor

Use the `WITH HOLD` keywords to create a *hold cursor*. A hold cursor allows uninterrupted access to a set of rows across multiple transactions.

Ordinarily, all cursors close at the end of a transaction. A hold cursor does not close; it remains open after a transaction ends.

A hold cursor can be either a sequential cursor or (in ESQL/C) a scroll cursor.

The `WITH HOLD` keywords are valid in SPL routines only for Select cursors. For the syntax of the DECLARE statement in SPL routines, see Declaring a Dynamic Cursor in an SPL Routine on page 449.

In ESQL/C, you can use the `WITH HOLD` keywords to declare Select and Function cursors (with the sequential attribute or the scroll attribute) and also to declare Insert cursors. These keywords follow the `CURSOR` keyword in the DECLARE statement. The following example creates a sequential hold cursor for a SELECT statement:

```
DECLARE hld_cur CURSOR WITH HOLD FOR
   SELECT customer_num, lname, city FROM customer;
```

You can use a select hold cursor as the following code example shows. This code fragment uses a hold cursor as a *master* cursor to scan one set of records and a sequential cursor as a *detail* cursor to point to records that are located in a different table. The records that the master cursor scans are the basis for updating the records to which the detail cursor points. The COMMIT WORK statement at the end of each iteration of the first WHILE loop leaves the hold cursor **c_master** open but closes the sequential cursor **c_detail** and releases all locks. This technique minimizes the resources that the database server must devote to locks and unfinished transactions, and it gives other users immediate access to updated rows.

```
EXEC SQL BEGIN DECLARE SECTION;
   int p_custnum, int save_status; long p_orddate;
EXEC SQL END DECLARE SECTION;

EXEC SQL prepare st_1 from
   'select order_date from orders where customer_num = ? for update';
EXEC SQL declare c_detail cursor for st_1;
EXEC SQL declare c_master cursor with hold for
   select customer_num from customer where city = 'Pittsburgh';

EXEC SQL open c_master;
if(SQLCODE==0) /* the open worked */
   EXEC SQL fetch c_master into :p_custnum; /* discover first customer */
while(SQLCODE==0) /* while no errors and not end of pittsburgh customers */
    {
   EXEC SQL begin work; /* start transaction for customer p_custnum */
   EXEC SQL open c_detail using :p_custnum;
   if(SQLCODE==0) /* detail open succeeded */
      EXEC SQL fetch c_detail into :p_orddate; /* get first order */
   while(SQLCODE==0) /* while no errors and not end of orders */
        {
      EXEC SQL update orders set order_date = '08/15/94'
         where current of c_detail;
      if(status==0) /* update was ok */
         EXEC SQL fetch c_detail into :p_orddate; /* next order */
      }
   if(SQLCODE==SQLNOTFOUND) /* correctly updated all found orders */
      EXEC SQL commit work; /* make updates permanent, set status */
   else /* some failure in an update */
        {
      save_status = SQLCODE; /* save error for loop control */
      EXEC SQL rollback work;
      SQLCODE = save_status; /* force loop to end */
       }
   if(SQLCODE==0) /* all updates, and the commit, worked ok */
      EXEC SQL fetch c_master into :p_custnum; /* next customer? */
   }
EXEC SQL close c_master;
```

Use either the CLOSE statement to close the hold cursor explicitly or the CLOSE DATABASE or DISCONNECT statements to close it implicitly. The CLOSE DATABASE statement closes all cursors.

Releases earlier than Version 9.40 of Informix® do not support the PDQPRIORITY feature with cursors that were declared WITH HOLD.

Using an Insert Cursor with Hold

If you associate a hold cursor with an INSERT statement, you can use transactions to break a long series of PUT statements into smaller sets of PUT statements. Instead of waiting for the PUT statements to fill the buffer and cause an automatic write to the database, you can execute a COMMIT WORK statement to flush the row buffer. With a hold cursor, COMMIT WORK commits the inserted rows but leaves the cursor open for further inserts. This method can be desirable when you are inserting a large number of rows, because pending uncommitted work consumes database server resources.

Subset of SELECT statements associated with cursors

As indicated in the syntax diagram for the DECLARE statement, not all SELECT statements can be associated with a read-only or update cursor.

If the DECLARE statement on page 433 includes the FOR READ ONLY or FOR UPDATE option, you must observe certain restrictions on the SELECT statement that is included in the DECLARE statement (either directly or as a prepared statement).

If the DECLARE statement includes the FOR READ ONLY option, the SELECT statement cannot have a FOR READ ONLY or FOR UPDATE option. (For a description of SELECT syntax and usage, see SELECT statement on page 690.)

If the DECLARE statement includes the FOR UPDATE option, the SELECT statement must conform to the following restrictions:

- The statement can select data from only one table.
- The statement cannot include any aggregate functions.
- The statement cannot also include the FOR UPDATE keywords.
- The statement cannot include any of the following clauses or keywords: DISTINCT, EXCEPT, FOR READ ONLY, FOR UPDATE, GROUP BY, INTERSECT, INTO TEMP, MINUS, ORDER BY, UNION, or UNIQUE.

Examples of Cursors in Non-ANSI Compliant Databases

In a database that is not ANSI compliant, a cursor associated with a SELECT statement is a read-only cursor by default. The following example declares a read-only cursor in a non-ANSI compliant database:

```
EXEC SQL declare cust_curs cursor for
   select * from customer_notansi;
```

If you want to make it clear in the program code that this cursor is a read-only cursor, specify the FOR READ ONLY option as the following example shows:

```
EXEC SQL declare cust_curs cursor for
   select * from customer_notansi for read only;
```

If you want this cursor to be an update cursor, specify the FOR UPDATE option in your DECLARE statement. This example declares an update cursor:

```
EXEC SQL declare new_curs cursor for
   select * from customer_notansi for update;
```

If you want an update cursor to be able to modify only some columns in a table, you must specify those columns in the FOR UPDATE clause. The following example declares an update cursor that can update only the **fname** and **lname** columns in the **customer_notansi** table:

```
EXEC SQL declare name_curs cursor for
   select * from customer_notansi for update of fname, lname;
```

Examples of Cursors in ANSI-Compliant Databases

In an ANSI-compliant database, a cursor associated with a SELECT statement is an update cursor by default.

The following example declares an update cursor in an ANSI-compliant database:

```
EXEC SQL declare x_curs cursor for select * from customer_ansi;
```

To make it clear in the program documentation that this cursor is an update cursor, you can specify the FOR UPDATE option as in this example:

```
EXEC SQL declare x_curs cursor for
   select * from customer_ansi for update;
```

If you want an update cursor to be able to modify only some of the columns in a table, you must specify these columns in the FOR UPDATE option. The following example declares an update cursor and specifies that this cursor can update only the **fname** and **lname** columns in the **customer_ansi** table:

```
EXEC SQL declare y_curs cursor for
   select * from customer_ansi for update of fname, lname;
```

If you want a cursor to be a read-only cursor, you must override the default behavior of the DECLARE statement by specifying the FOR READ ONLY option in your DECLARE statement. The following example declares a read-only cursor:

```
EXEC SQL declare z_curs cursor for
   select * from customer_ansi for read only;
```

Associating a Cursor with a Prepared Statement

The PREPARE statement lets you assemble the text of an SQL statement at runtime and pass the statement text to the database server for execution. If you anticipate that a dynamically prepared SELECT, EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement that returns values could produce more than one row of data, the prepared statement must be associated with a cursor. (See PREPARE.)

The result of a PREPARE statement is a statement identifier (*statement id* or *id variable*), which is a data structure that represents the prepared statement text. To declare a cursor for the statement text, associate a cursor with the statement identifier.

You can associate a sequential cursor with any prepared SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement. You cannot associate a scroll cursor with a prepared INSERT statement or with a SELECT statement that was prepared to include a FOR UPDATE clause.

After a cursor is opened, used, and closed, a different statement can be prepared under the same statement identifier. In this way, it is possible to use a single cursor with different statements at different times. The cursor must be redeclared before you use it again.

The following example contains code that prepares a SELECT statement and declares a sequential cursor for the prepared statement text. The statement identifier **st_1** is first prepared from a SELECT statement that returns values; then the cursor **c_detail** is declared for **st_1**.

```
EXEC SQL prepare st_1 from
   'select order_date
      from orders where customer_num = ?';
EXEC SQL declare c_detail cursor for st_1;
```

If you want to use a prepared SELECT statement to modify data, add a FOR UPDATE clause to the statement text that you want to prepare, as the following example shows:

```
EXEC SQL prepare sel_1 from
   'select * from customer for update';
EXEC SQL declare sel_curs cursor for sel_1;
```

DDL operations that change the schema of a table can invalidate a cursor whose associated prepared statement or associated routine references the modified table, unless the prepared objects are recompiled, or unless the routine is reoptimized. For more information, see the section DDL Operations on Tables Referenced by Cursors on page 622.

The DECLARE statement allows you to declare a cursor for the collection variable. Such a cursor is called a *Collection cursor*. You use a collection variable to access the elements of a collection (SET, MULTISET, LIST) column. Use a cursor when you want to access one or more elements in a collection variable.

The Collection-Derived Table segment identifies the collection variable for which to declare the cursor. For more information, see Collection-Derived Table on page 1212.

Select with a Collection-Derived Table

The diagram for DECLARE statement on page 433 refers to this section.

To declare a Select cursor for a collection variable, include the Collection- Derived Table segment with the SELECT statement that you associate with the Collection cursor. A Select cursor allows you to select one or more elements *from* the collection variable. (For a description of SELECT syntax and usage, see SELECT statement on page 690.)

When you declare a Select cursor for a collection variable, the DECLARE statement has the following restrictions:

- It cannot include the FOR READ ONLY keywords as cursor mode.

  The Select cursor is an update cursor.
- It cannot include the SCROLL or WITH HOLD keywords.

  The Select cursor must be a sequential cursor.

In addition, the SELECT statement that you associate with the collection cursor has the following restrictions:

- It cannot include the following clauses or options: WHERE, GROUP BY, ORDER BY, HAVING, INTO TEMP, and WITH REOPTIMIZATION.
- It cannot contain expressions in the projection list.
- If the collection contains elements of opaque, distinct, built-in, or other collection data types, the projection list must be an asterisk (*).
- Column names in the projection list must be simple column names, without qualifiers.

  These columns cannot use the following syntax:

  ```
  database@server:table.column --INVALID SYNTAX
  ```

- It *must* specify the name of the collection variable in the FROM clause.

  You cannot specify an input parameter (the question-mark (?) symbol) for the collection variable. Likewise, you cannot use the virtual table format of the Collection-Derived Table segment.

Using a Select Cursor with a Collection Variable

About this task

A Collection cursor that includes a SELECT statement with the Collection- Derived Table clause provides access to the elements in a collection variable.

To select more than one element:

1. Create a client collection variable in your program.
2. Declare the Collection cursor for the SELECT statement with the DECLARE statement.
   To modify elements of the collection variable, declare the Select cursor as an update cursor with the FOR UPDATE keywords. You can then use the WHERE CURRENT OF clause of the DELETE and UPDATE statements to delete or update elements of the collection.
3. Open this cursor with the OPEN statement.
4. Fetch the elements from the Collection cursor with the FETCH statement and the INTO clause.
5. If necessary, perform any updates or deletes on the fetched data and save the modified collection variable in the collection column.
   Once the collection variable contains the correct elements, use the UPDATE or INSERT statement to save the contents of the collection variable in the actual collection column (SET, MULTISET, or LIST).
6. Close the Collection cursor with the CLOSE statement.

Results

This DECLARE statement declares a Select cursor for a collection variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(integer not null) a_set;
EXEC SQL END DECLARE SECTION;
```

```
...
EXEC SQL declare set_curs cursor for select * from table(:a_set);
```

For an extended code example that uses a Collection cursor for a SELECT statement, see Fetching from a Collection Cursor on page 520.

Insert with a Collection-Derived Table

About this task

To declare an Insert cursor for a collection variable, include the Collection- Derived Table segment in the INSERT statement associated with the Collection cursor. An Insert cursor can insert one or more elements in the collection. For a description of INSERT syntax and usage, see INSERT statement on page 580.

The Insert cursor must be a sequential cursor. That is, the DECLARE statement cannot specify the SCROLL keyword.

When you declare an Insert cursor for a collection variable, the Collection- Derived Table clause of the INSERT statement *must* contain the name of the collection variable. You cannot specify an input parameter (the question-mark (?) symbol) for the collection variable. However, you can use an input parameter in the VALUES clause of the INSERT statement. This parameter indicates that the collection element is to be provided later by the FROM clause of the PUT statement.

A Collection cursor that includes an INSERT statement with the Collection- Derived Table clause allows you to insert more than one element into a collection variable.

To insert more than one element:

1. Create a client collection variable in your program.
2. Declare the Collection cursor for the INSERT statement with the DECLARE statement.
3. Open the cursor with the OPEN statement.
4. Put the elements into the Collection cursor with the PUT statement and the FROM clause.
5. Once the collection variable contains all the elements, use the UPDATE statement or the INSERT statement on a table name to save the contents of the collection variable in a collection column (SET, MULTISET, or LIST).
6. Close the Collection cursor with the CLOSE statement.

Results

This example declares an Insert cursor for the **a_set** collection variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection multiset(smallint not null) a_mset;
   int an_element;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL declare mset_curs cursor for
   insert into table(:a_mset) values (?);
EXEC SQL open mset_curs;
while (1)
{
...
   EXEC SQL put mset_curs from :an_element;
```

```
...
}
```

To insert the elements into the collection variable, use the PUT statement with the FROM clause. For a code example that uses a Collection cursor for an INSERT statement, see Inserting into a Collection Cursor on page 639.

Using Cursors with Transactions

To roll back a modification, you must perform the modification within a transaction. A transaction in a database that is not ANSI compliant begins only when the BEGIN WORK statement is executed.

In an ANSI-compliant database, transactions are always in effect.

The database server enforces these guidelines for select and update cursors to ensure that modifications can be committed or rolled back properly:

- Open an insert or update cursor within a transaction.
- Include PUT and FLUSH statements within one transaction.
- Modify data (update, insert, or delete) within one transaction.

The database server lets you open and close a hold cursor for an update outside a transaction; however, you should fetch all the rows that pertain to a given modification and then perform the modification all within a single transaction. You cannot open and close a hold cursor or an update cursor outside a transaction.

The following example uses an update cursor within a transaction:

```
EXEC SQL declare q_curs cursor for
   select customer_num, fname, lname from customer
   where lname matches :last_name for update;
EXEC SQL open q_curs;
EXEC SQL begin work;
EXEC SQL fetch q_curs into :cust_rec; /* fetch after begin */
EXEC SQL update customer set lname = 'Smith'
   where current of q_curs;
/* no error */
EXEC SQL commit work;
```

When you update a row within a transaction, the row remains locked until the cursor is closed or the transaction is committed or rolled back. If you update a row when no transaction is in effect, the row lock is released when the modified row is written to disk. If you update or delete a row outside a transaction, you cannot roll back the operation.

In a database that uses transactions, you cannot open an Insert cursor outside a transaction unless it was also declared with the WITH HOLD keywords.

Declaring a Dynamic Cursor in an SPL Routine

Use the DECLARE statement in an SPL routine to declare the name of a dynamic cursor, and to associate that cursor with the statement identifier of a prepared statement that the PREPARE statement has declared in the same SPL routine.

Dynamic cursors that the DECLARE statement of SQL can create in SPL routines are distinct from the direct sequential cursors that the FOREACH statement of SPL can create in SPL routines. (For the syntax and usage of direct sequential cursors, see FOREACH on page 935.)

Syntax

The syntax of the DECLARE statement in SPL routines is a subset of the syntax that DECLARE supports in routines.

(explicit id unique_597_Connect_42_declspl) unique_597_Connect_42_declspl



Element	Description	Restrictions	Syntax
cursor_id	Name declared here for a new dynamic cursor	Must be unique among names of cursors, prepared statements, and SPL variables in the routine	Identifier on page 1231
statement_id	Identifier of a single prepared SQL statement	Must have been declared in a PREPARE statement of the same SPL routine	Identifier on page 1231

Usage

In UDRs written in the SPL language, the *statement_id* associated with the cursor must have been declared earlier in the same UDR by a PREPARE statement from the text of a single SQL statement of one of these statement types:

- EXECUTE FUNCTION
- EXECUTE PROCEDURE
- SELECT.

This prepared statement text that *statement_id* specifies can include question mark (?) symbols as placeholders for values that the user supplies at runtime, but these placeholders in the PREPARE statement can represent only data values, not SQL identifiers.

Dynamic cursors that the DECLARE statement can define in SPL routines resemble ESQL/C Select cursors or Function cursors in their functionality, but with these restrictions:

- Cursors that DECLARE defines in an SPL routine can be Select cursors or Function cursors, but they cannot be Insert cursors nor Collection cursors.
- The identifier of the cursor or of the prepared statement cannot be specified as an SPL variable, because in SPL, the identifiers of variables, cursors, and prepared objects all share the same namespace.
- By default, dynamic cursors of SPL are sequential. They cannot be scroll cursors.
- The semantics of dynamic cursors that you create with the WITH HOLD keywords are the same as for hold cursors that the FOREACH statement declares.
- The WITH HOLD keywords are valid in SPL routines only for Select cursors. If *statement_id* references the prepared text of an EXECUTE FUNCTION or EXECUTE PROCEDURE statement, the DECLARE statement fails with error -26056.
- The FOR UPDATE and FOR READ ONLY keywords that ESQL/C supports in DECLARE statements are not supported in SPL routines. Use the FOREACH statement of SPL to declare direct cursors that can emulate the functionality of

ESQL/C update cursors. (But queries associated with direct cursors are defined when the UDR is compiled, rather than at runtime.)

- The DECLARE statement in SPL routines does not support SELECT operations on collection-derived tables.
- Syntax errors in DECLARE statements of SPL routines are reported at runtime, unlike syntax errors of ESQL/C, which are reported when the routine is compiled.

The names of the dynamic cursors that the DECLARE statement associates with a prepared statement in SPL routines can be referenced by the OPEN, CLOSE, FETCH, and FREE statements of dynamic SQL in the same SPL routine.

In the following program fragment, a cursor called **equi_noctis** is declared, opened, closed, and freed.

```
CREATE FUNCTION lente
   DEFINE first, last VARCHAR(30);
   . . .
   DATABASE stores_demo;
   LET first = "SELECT * FROM state";
   LET lsst = "WHERE code < ?";
   PREPARE stmt_1 FROM first || last;
   DECLARE equi_noctis FOR stmt_1;
   OPEN equi_noctis
   . . .
   CLOSE equi_noctis;
   FREE equi_noctis;
   FREE stmt_1;
   ...
   END FUNCTION;
```

DELETE statement

Use the DELETE statement to delete one or more rows from a table, or to delete one or more elements from a collection variable of SPL or of .

Syntax

(explicit id unique_559_Connect_42_del002) unique_559_Connect_42_del002(explicit id unique_559_Connect_42_del003)

unique_559_Connect_42_del003(explicit id unique_559_Connect_42_del004) unique_559_Connect_42_del004(explicit id unique_559_Connect_42_del005)

unique_559_Connect_42_del005(explicit id unique_559_Connect_42_del007) unique_559_Connect_42_del007(explicit id unique_559_Connect_42_del008)

unique_559_Connect_42_del008(explicit id unique_559_Connect_42_del009) unique_559_Connect_42_del009(explicit id unique_559_Connect_42_del010)

unique_559_Connect_42_del010 **DELETE** [<Optimizer Directives>] { **FROM** | } { { *table* | *view* | *synonym* } [**AS**] *alias*] | { | { **ONLY** ({ *table* | *synonym* }) | <Collection-Derived Table> } } } [{ **WHERE***condition* | **WHERE CURRENT OF** *cursor_id* }]

Element	Description	Restrictions	Syntax
alias	Temporary name declared here for a table, view, or synonym.	The AS keyword must precede *alias* if WHERE is the identifier of *alias*	Identifier on page 1231
condition	Logical criteria that deleted rows must satisfy	Cannot be a UDR nor a correlated subquery	Condition on page 967

Element	Description	Restrictions	Syntax
cursor_id	Previously declared cursor	Must have been declared FOR UPDATE	Identifier on page 1231
synonym, *table*, *view*	Synonym, table, or updatable view with rows to be deleted	The *table* or *view* (or *synonym* and the table or view to which it points) must exist	Database Object Name on page 1225

Usage

Use the DELETE statement to remove any of the following types of database objects or program objects:

- A row in a table or in a view: a single row, a group of rows, or all rows
- An element in a column of a collection data type
- In a column of a named or unnamed ROW data type, a field, or all fields.

You can also use this statement to remove the values in one or more elements in or SPL collection variables or ROW variables.

To execute the DELETE statement, you must hold the DBA access privilege on the database, or the Delete access privilege on the table.

If you specify a *view* name, the view must be updatable. For an explanation of an updatable view, see Updating Through Views on page 426.

The DELETE statement cannot reference table objects that the CREATE EXTERNAL TABLE statement defined.

In a database with explicit transaction logging, any DELETE statement that you execute outside a transaction is treated as a single transaction.

In an ANSI-compliant database, data manipulation language (DML) statements are always in a transaction. You cannot execute a DELETE statement outside a transaction.

FROM clause

The `FROM` keyword that precedes the name of the target table is optional. To delete rows from a table named from, you can set the **DELIMIDENT** environment variable and use double quotation marks (`"`) to delimit "from":

```
DELETE "from";
```

Alternatively, you can qualify the name of the from table with the name of its owner:

```
DELETE zelaine.from;
```

Your SQL code will be easier for humans to read and to maintain, however, if you avoid declaring SQL keywords as identifiers of tables, views, or other database objects.

WHERE clause

If you use DELETE without a WHERE clause (to specify either a condition or the active set of the cursor), all rows in the table are deleted:

```
DELETE FROM tableZ;
```

It is typically more efficient, however, to use the TRUNCATE statement, rather than the DELETE statement, to remove all rows from a table.

In DB-Access, if you omit the WHERE clause while working at the SQL menu, prompts you to verify that you want to delete all rows from a table. You do not receive a prompt if you execute DELETE within a command file.

Locking considerations

The database server locks each row affected by a DELETE statement within a transaction for the duration of the transaction. The locking granularity of the table can be PAGE level or ROW level.

Features that determine the locking granularity have this ascending order of precedence:

- The DEF_TABLE_LOCKMODE configuration parameter can set the default granularity for table locks to PAGE or to ROW.
- If the IFX_TABLE_LOCKMODE environment variable is set to PAGE or ROW, its setting overrides any DEF_TABLE_LOCKMODE default.
- The LOCK MODE clause of the CREATE TABLE statement overrides any default locking granularity for the new table.
- The LOCK MODE clause of the ALTER TABLE statement can reset the locking granularity to PAGE or to ROW for a table, overriding any of the settings above.
- The LOCK TABLE statement always locks the entire table, overriding any locking granularity specification listed above for the specified table.

When the locking granularity of the table is ROW, the database server acquires one lock for each row affected by the DELETE statement.

If the number of rows affected is very large, and the lock mode is ROW, you might exceed the limits that your operating system places on the maximum number of simultaneous locks. If this occurs, you can either reduce the scope of the DELETE statement, or you can use LOCK TABLE statement to lock the table in exclusive mode before you execute the DELETE statement.

Using the ONLY keyword with typed tables

When the DELETE statement specifies a supertable, any qualifying rows that satisfy the WHERE clause are also deleted, by default, from all the subtables of the supertable within the table hierarchy. To restrict the scope of DELETE to the supertable, you must specify the `ONLY` keyword before the name or synonym of the supertable.

In the following example, any rows whose name column has the value **johnson** will be deleted from the **super_tab** supertable. Any rows with **johnson** in their name column will persist, however, in the subtables of **super_tab**, because the `ONLY( )` clause restricts the DELETE operation to that supertable:

```
DELETE FROM ONLY(super_tab)   -- scope excludes child tables
   WHERE name = "johnson";
```

Warning: If you use the DELETE statement on a supertable and omit the ONLY keyword and the WHERE clause, all rows of the supertable and of all its subtables are deleted, as in the following example.

```
DELETE FROM super_tab;   -- deletes all rpws in the  hierarchy
```

You cannot specify the ONLY keyword if you plan to use the WHERE CURRENT OF clause to delete the current row of the active set of a cursor.

Considerations When Tables Have Cascading Deletes

When you use the ON DELETE CASCADE option of the REFERENCES clause of either the CREATE TABLE or ALTER TABLE statement, you specify that you want deletes to cascade from one table to another. For example, in the **stores_demo** database**,** the **stock** table contains the column **stock_num** as a primary key. The **catalog** and **items** tables each contain the column **stock_num** as foreign keys with the ON DELETE CASCADE option specified. When a delete is performed from the **stock** table, rows are also deleted in the **catalog** and **items** tables, which are referenced through the foreign keys.

To have DELETE actions cascade to a table that has a referential constraint on a parent table, you need the Delete privilege only on the parent table that you reference in the DELETE statement.

If a DELETE operation with no WHERE clause is performed on a table that one or more child tables reference with cascading deletes, Informix® deletes all rows from that table and from any affected child tables. (This resembles the effect of the TRUNCATE statement, but Informix® does not support TRUNCATE operations on any table that has a child table referencing it.)

For an example of how to create a referential constraint that uses cascading deletes, see Using the ON DELETE CASCADE Option on page 334.

Restrictions on DELETE When Tables Have Cascading Deletes

You cannot use a child table in a correlated subquery to delete a row from a parent table. If two child tables reference the same parent table, and one child specifies cascading deletes but the other child does not, then if you attempt to delete a row that applies to both child tables from the parent table, the delete fails, and no rows are deleted from the parent or child tables.

Locking and Logging Implications of Cascading Deletes

During deletes, the database server places locks on all qualifying rows of the referenced and referencing tables.

Informix® requires transaction logging for cascading deletes. If logging is turned off in a database that is not ANSI-compliant, even temporarily, deletes do not cascade, because you cannot roll back any actions. For example, if a parent row is deleted, but the system fails before the child rows are deleted, the database will have dangling child records, in violation of referential integrity. After logging is turned back on, however, subsequent deletes cascade.

Subqueries in the WHERE Clause of DELETE

The FROM clause of a subquery in the WHERE clause of the DELETE statement can specify as a data source the same table or view that the FROM clause of the DELETE statement specifies. DELETE operations with subqueries that reference the same table object are supported only if all of the following conditions are true:

- The subquery either returns a single row, or else has no correlated column references.
- The subquery is in the DELETE statement WHERE clause, using Condition with Subquery syntax.
- Any SPL routines within the subquery cannot reference the table that is being modified.

Unless all of these conditions are satisfied, DELETE statements that include subqueries that reference the same table or view that the DELETE statement modifies return error -360.

The following example deletes from the **orders** table a subset of rows whose **paid_date** column value satisfies the condition in the WHERE clause. The WHERE clause specifies which rows to delete by applying the IN operator to the rows returned by a subquery that selects only the rows of the **orders** table where the **paid_date** value is earlier than the current date:

```
DELETE FROM orders WHERE paid_date IN
   (SELECT paid_date FROM orders WHERE paid_date < CURRENT );
```

This subquery includes only uncorrelated column references, because its only referenced column is in a table specified in its FROM clause. The requirements listed above are in effect, because the data source of the subquery is the same orders table that the FROM clause of the outer UPDATE statement specifies. The previous example illustrates Informix® support for uncorrelated subqueries in the WHERE clause of the DELETE statement. rather than how to write short SQL statements. The next example achieves the same result with simpler syntax:

```
DELETE orders WHERE paid_date < CURRENT;
```

The following example deletes from the **stock** table the row (or rows) with the largest **unit_price** value. The WHERE clause identifies which **unit_price** value is the largest by applying the equality operator to the result of a subquery that calls the built-in **MAX** aggregate function for the **unit_price** column values:

```
DELETE FROM stock WHERE unit_price =
   (SELECT MAX(unit_price) FROM stock );
```

If an enabled Select trigger is defined on a table that is the data source of a subquery in the WHERE clause of a DELETE statement that modifies the same table, executing that subquery within the DELETE statement does not activate the trigger.

A subquery in the DELETE statement can include the UNION or UNION ALL operators.

If the table that the outer DELETE statement modifies a typed table within a table hierarchy, Informix® supports all of the following operations that use valid subqueries in the WHERE clause of DELETE:

- DELETE from parent table with subquery (SELECT from parent table)
- DELETE from parent table with subquery (SELECT from child table)
- DELETE from child table with subquery (SELECT from parent table)
- DELETE from child table with subquery (SELECT from child table).

See the Condition with Subquery on page 979 topic for more information about the syntax of subqueries to return multiple rows as predicates in the WHERE clause of the DELETE statement.

Declaring an alias for the table

You can declare an alias for the table. The alias can reference the fully-qualified database object name of a local or remote table, view, or synonym.

The alias is a temporary name that is not registered in the system catalog of the database, and that persists only while the DELETE statement is running.

If the name that you declare as the alias is the keyword WHERE, you must use the AS keyword to clarify the syntax:

```
DELETE stock AS where
   WHERE manu_code =
      (SELECT manu_code FROM where WHERE manu_code MATCHES 'H*');
```

Because **where** is a keyword of both the DELETE and SELECT statements, the previous example is not easy to read. The following example accesses a remote table without declaring an alias for the table:

```
DELETE overstock@cleveland:stock AS ocs
   WHERE manu_code =
      (SELECT manu_code FROM overstock@cleveland:stock
       WHERE manu_code MATCHES 'H*');
;
```

The next example is logically equivalent to the previous DELETE statement, but simplifies the notation by declaring **ocs** as alias that references the same table in the subquery"

```
DELETE overstock@cleveland:stock AS ocs
   WHERE manu_code =
      (SELECT manu_code FROM ocs WHERE manu_code MATCHES 'H*');
;
```

Using the WHERE CURRENT OF Keywords (ESQL/C, SPL)

The WHERE CURRENT OF clause deletes the current row of the active set of a cursor. When you include this clause, the DELETE statement removes the row of the active set at the current position of the cursor. After the deletion, no current row exists; you cannot use the cursor to delete or update a row until you reposition the cursor with a FETCH statement (in ESQL/C routines) or with a FOREACH statement (in SPL routines).

You access the current row of the active set of a cursor with an update cursor. Before you can use the WHERE CURRENT OF clause, you must first create an update cursor by using the FOREACH statement (in SPL) or the DECLARE statement with the FOR UPDATE clause (in). The *cursor_id* that follows the OF keyword cannot be declared, however, by the DECLARE statement in an SPL routine

Unless they are declared with the FOR READ ONLY keywords, all Select cursors are potentially update cursors in an ANSI-compliant database. You can use the WHERE CURRENT OF clause with any Select cursor that was not declared with the FOR READ ONLY keywords.

You cannot use WHERE CURRENT OF if you are selecting from only one table in a table hierarchy. That is, this clause is not valid with the ONLY keyword.

The WHERE CURRENT OF clause can be used to delete an element from a collection by deleting the current row of the collection-derived table that a collection variable holds. For more information, see Collection-Derived Table on page 1212.

Deleting Rows That Contain Opaque Data Types

Some opaque data types require special processing when they are deleted. For example, if an opaque data type contains spatial or multirepresentational data, it might provide a choice of how to store the data: inside the internal structure or, for large objects, in a smart large object.

To accomplish this process, call a user-defined support function called **destroy( )**. When you use DELETE to remove a row that contains one of these opaque types, the database server automatically invokes **destroy( )** for the opaque type. This function decides how to remove the data, regardless of where it is stored. For more information on the **destroy( )** support function, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

Deleting Rows That Contain Collection Data Types

When a row contains a column that is a collection data type (LIST, MULTISET, or SET), you can search for a particular element in the collection, and delete the row or rows in which the element is found.

For example, the following statement deletes any rows from the **new_tab** table in which the **set_col** column contains the element `jimmy smith`:

```
DELETE FROM new_tab WHERE 'jimmy smith' IN set_col;
```

You can also use a collection variable to delete values in a collection column by deleting one or more individual elements in a collection. For more information, see Collection-Derived Table on page 1212 and the examples in Database Name on page 1224 and Example of Deleting from a Collection on page 1221.

Data Types in Distributed DELETE Operations

Distributed DELETE operations across tables in databases of different server instances can return only a subset of the data types that distributed DELETE operations can return from tables that are all in databases of the same Informix® instance.

A DELETE statement (or any other SQL data-manipulation language statement) that accesses a database of another Informix® instance can reference only the following data types:

- Built-in data types that are not opaque or complex
- BOOLEAN
- BSON
- JSON
- LVARCHAR
- DISTINCT of built-in types that are not opaque
- DISTINCT of BOOLEAN
- DISTINCT of BSON

- DISTINCT of JSON
- DISTINCT of LVARCHAR
- DISTINCT of the DISTINCT types in this list.

Cross-server distributed DELETE operations can support these DISTINCT types only if the DISTINCT types are cast explicitly to built-in types, and all of the DISTINCT types, their data type hierarchies, and their casts are defined exactly the same way in each participating database.

Cross-server DML operations cannot reference a column or expression of a complex, large-object, nor user-defined opaque data type (UDT), nor of an unsupported DISTINCT type or built-in opaque type. For additional information about the data types that Informix® supports in cross-server DML operations, see Data Types in Cross-Server Transactions on page 702.

Distributed operations that access other databases of the local Informix® instance, however, can access the same data types that are listed above for cross-server operations, and also the following additional data types:

- Most of the *built-in opaque data types*, as listed in Data Types in Cross-Database Transactions on page 700
- DISTINCT of the built-in types that are referenced in the line above
- DISTINCT of any of the data types that are listed in either of the two lines above
- Opaque user-defined data types (UDTs) that can be cast explicitly to built-in data types.

Cross-database DELETE operations can support these DISTINCT and opaque UDTs only if all the opaque and DISTINCT UDTs are cast explicitly to built-in types, and all of the opaque UDTs, DISTINCT types, data type hierarchies, and casts are defined exactly the same way in each of the participating databases.

Distributed DELETE statements cannot access a database of another Informix® instance unless both server instances support either a TCP/IP or an IPCSTR connection, as defined in their DBSERVERNAME or DBSERVERALIASES configuration parameters and in the **sqlhosts** information. This connection-type requirement applies to any communication between Informix® instances, even if both database servers reside on the same computer.

SQLSTATE Values in an ANSI-Compliant Database

If no rows satisfy the WHERE clause of a DELETE operation on a table in an ANSI-compliant database, the database server issues a warning. You can detect this warning condition in either of the following ways:

- The GET DIAGNOSTICS statement sets the **RETURNED_SQLSTATE** field to the value `02000`. In an SQL API application, the **SQLSTATE** variable contains this same value.
- In an SQL API application, the **sqlca.sqlcode** and **SQLCODE** variables contain the value `100`.

The database server also sets **SQLSTATE** and **SQLCODE** to these values if the DELETE . . . WHERE statement is part of a multistatement prepared object, and the database server returns no rows.

SQLSTATE Values in a Database That Is Not ANSI-Compliant

In a database that is not ANSI compliant, the database server does not return a warning when it finds no rows satisfying the WHERE clause of a DELETE statement. In this case, the **SQLSTATE** code is `00000` and the **SQLCODE** code is zero (`0`). If the

DELETE . . . WHERE is part of a multistatement prepared object, however, and no rows are returned, the database server does issue a warning. It sets **SQLSTATE** to `02000` and sets the **SQLCODE** value to `100`.

For information on the ANSI/ISO-compliance status of values returned to the **SQLSTATE** variable, see the section SQLSTATE Support for the ANSI/ISO Standard for SQL on page 527.

DESCRIBE statement

Use the DESCRIBE statement to obtain information about output parameters and other features of a prepared object before you execute it.

Use this statement with . (See also DESCRIBE INPUT statement on page 463.)

Syntax

DESCRIBE [**OUTPUT**] { *statement_id_var* | *statement_id* } { **USING SQL DESCRIPTOR** { *descriptor_var* | *'descriptor'* } | **INTO** { **SQL DESCRIPTOR** { *descriptor_var* | *'descriptor'* } | *sqlda_pointer* } }

Element	Description	Restrictions	Syntax
descriptor	Name of a system-descriptor area	System-descriptor area must already be allocated	Quoted String on page 1202
descriptor_var	Host variable specifying a system-descriptor area	Must contain the name of an allocated system-descriptor area	Language-specific rules for names
sqlda_pointer	Pointer to an **sqlda** structure	Cannot begin with dollar ($) sign or colon (:). An **sqlda** structure is required if dynamic SQL is used.	See the **sqlda** structure in the *HCL® Informix® Enterprise Replication Guide*
statement_id	Statement identifier for a prepared SQL statement	Must be defined in a previous PREPARE statement	PREPARE statement on page 623; Identifier on page 1231
statement_id_var	Host variable that contains the value of *statement_id*	Must be declared in a previous PREPARE statement	Language-specific rules for names

Usage

DESCRIBE can provide information at runtime about a prepared statement:

- The type of SQL statement that was prepared
- Whether an UPDATE or DELETE statement contains a WHERE clause
- For an EXECUTE, EXECUTE FUNCTION, EXECUTE PROCEDURE, INSERT, SELECT, or UPDATE statement, the DESCRIBE statement also returns the number, data types, and size of the output parameter values.
- For a SELECT statement, DESCRIBE also returns the name of the column or expression that the query returns.

With this information, you can write code to allocate memory to hold retrieved values and display or process them after they are fetched.

The OUTPUT Keyword

The OUTPUT keyword specifies that only information about output parameters of the prepared statement are stored in the **sqlda** descriptor area. If you omit this keyword, DESCRIBE can return input parameters, but only for INSERT statements (and for UPDATE, if the **IFX_UPDDESC** environment variable is set in the environment where the database server is initialized).

Describing the Statement Type

The DESCRIBE statement takes a statement identifier from a PREPARE statement as input. When the DESCRIBE statement executes, the database server sets the value of the SQLCODE field of the **sqlca** to indicate the statement type (that is, the keyword with which the statement begins). If the prepared statement text contains more than one SQL statement, the DESCRIBE statement returns the type of the first statement in the text.

SQLCODE is set to zero to indicate a SELECT statement *without* an INTO TEMP clause. This situation is the most common. For any other SQL statement, SQLCODE is set to a positive integer. You can test the number against the constant names that are defined. In , the constant names are defined in the **sqlstypes.h** header file.

The DESCRIBE statement (and the DESCRIBE INPUT statement) use the SQLCODE field differently from any other statement, possibly returning a nonzero value when it executes successfully. You can revise standard error-checking routines to accommodate this behavior, if desired.

Checking for the Existence of a WHERE Clause

If the DESCRIBE statement detects that a prepared statement contains an UPDATE or DELETE statement without a WHERE clause, the DESCRIBE statement sets the **sqlca.sqlwarn.sqlwarn4** variable to `w`.

When you do not specify a WHERE clause in either a DELETE or UPDATE statement, the database server performs the delete or update operation on the entire table. Check the **sqlca.sqlwarn.sqlwarn4** variable to avoid unintended global changes to your table.

Describing a Statement with Runtime Parameters

If the prepared statement contains parameters for which the number of parameters or parameter data types is to be supplied at runtime, you can describe these input values. If the prepared statement text includes one of the following statements, the DESCRIBE statement returns a description of each column or expression that is included in the list:

- EXECUTE FUNCTION (or EXECUTE PROCEDURE)
- INSERT
- SELECT (without an INTO TEMP clause)
- UPDATE

  In Informix®, the **IFX_UPDDESC** environment variable, as described in the *HCL® Informix® Guide to SQL: Reference*, must be set before you can use DESCRIBE to obtain information about an UPDATE statement.

The description includes the following information:

- The data type of the column, as defined in the table
- The length of the column, in bytes
- The name of the column or expression

For a prepared INSERT or UPDATE statement, DESCRIBE returns only the dynamic parameters (those expressed with a question mark (?) symbol). Using the OUTPUT keyword, however, prevents these from being returned.

You can specify a destination for the returned information as a new or existing system-descriptor area, or as a pointer to an **sqlda** structure.

A system-descriptor area conforms to the X/Open standards.

Using the SQL DESCRIPTOR Keywords

Use the USING SQL DESCRIPTOR clause to store the description of a prepared statement list in a previously allocated system-descriptor area.

Use the INTO SQL DESCRIPTOR clause to create a new system-descriptor structure and store the description of a statement list in that structure.

To describe one of the previously mentioned statements into a system-descriptor area, DESCRIBE updates the system-descriptor area in these ways:

- It sets the COUNT field in the system-descriptor area to the number of values in the statement list. An error results if COUNT is greater than the number of item descriptors in the system-descriptor area.
- It sets the TYPE, LENGTH, NAME, SCALE, PRECISION, and NULLABLE fields in the system-descriptor area.

  If the column has an opaque data type, the database server sets the EXTYPEID, EXTYPENAME, EXTYPELENGTH, EXTYPEOWNERLENGTH, and EXTYPEOWNERNAME fields of the item descriptor.
- It allocates memory for the DATA field for each item descriptor, based on the TYPE and LENGTH information.

After a DESCRIBE statement is executed, the SCALE and PRECISION fields contain the scale and precision of the column, respectively. If SCALE and PRECISION are set in the SET DESCRIPTOR statement, and TYPE is set to DECIMAL or MONEY, the LENGTH field is modified to adjust for the scale and precision of the decimal value. If TYPE is not set to DECIMAL or MONEY, the values for SCALE and PRECISION are not set, and LENGTH is unaffected.

You must modify system-descriptor-area information with SET DESCRIPTOR statements to show the address in memory that is to receive the described value. You can change the data type to another compatible type. This change causes data conversion to take place when data values are fetched.

You can use the system-descriptor area in prepared statements that support a USING SQL DESCRIPTOR clause, such as EXECUTE, FETCH, OPEN, and PUT.

The following examples show the use of a system descriptor in a DESCRIBE statement. In the first example, the system descriptor is a quoted string; in the second example, it is an embedded variable name.

```
main()
{
. . .
EXEC SQL allocate descriptor 'desc1' with max 3;
EXEC SQL prepare curs1 FROM 'select * from tab';
EXEC SQL describe curs1 using sql descriptor 'desc1';
}
EXEC SQL describe curs1 using sql descriptor :desc1var;
```

Using the INTO sqlda Pointer Clause

Use the INTO *sqlda_pointer* clause to allocate memory for an **sqlda** structure and store its address in an **sqlda** pointer. The DESCRIBE statement fills in the allocated memory with descriptive information. Unlike the USING clause, the INTO clause creates new **sqlda** structures to store the output from DESCRIBE.

The DESCRIBE statement sets the **sqlda.sqld** field to the number of values in the statement list. The **sqlda** structure also contains an array of data descriptors (**sqlvar** structures), one for each value in the statement list. After a DESCRIBE statement is executed, the **sqlda.sqlvar** structure has the **sqltype**, **sqllen**, and **sqlname** fields set.

If the column has an opaque data type, DESCRIBE...INTO sets the **sqlxid**, **sqltypename**, **sqltypelen**, **sqlownerlen**, and **sqlownername** fields of the item descriptor.

The DESCRIBE statement allocates memory for an **sqlda** pointer once it is declared in a program. The application program, however, must designate the storage area of the **sqlda.sqlvar.sqldata**fields.

Describing a Collection Variable

The DESCRIBE statement can provide information about a collection variable when you use the USING SQL DESCRIPTOR or INTO clause. You must issue the DESCRIBE statement *after* you open the Select or Insert cursor, because the OPEN...USING statement specifies the name of the collection variable to use.

The next code fragment dynamically selects the elements of the **:a_set** collection variable into a system-descriptor area called **desc1**:

```
EXEC SQL BEGIN DECLARE SECTION;
      client collection a_set;
      int i, set_count;
      int element_type, element_value;
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate collection :a_set;
EXEC SQL allocate descriptor 'desc1';
EXEC SQL select set_col into :a_set from table1;
EXEC SQL prepare set_id from 'select * from table(?)'

EXEC SQL declare set_curs cursor for set_id;
EXEC SQL open set_curs using :a_set;
EXEC SQL describe set_id using sql descriptor 'desc1';
```

```
do
{
      EXEC SQL fetch set_curs using sql descriptor 'desc1';
      ...
      EXEC SQL get descriptor 'desc1' :set_count = count;
      for (i = 1; i <= set_count; i++)
      {
         EXEC SQL get descriptor 'desc1' value :i
            :element_type = TYPE;
         switch
         {
            case SQLINTEGER:
               EXEC SQL get descriptor 'desc1' value :i
                  :element_value = data;
            ...
         } /* end switch */
      } /* end for */
} while (SQLCODE == 0);

EXEC SQL close set_curs;
EXEC SQL free set_curs;
EXEC SQL free set_id;
EXEC SQL deallocate collection :a_set;
EXEC SQL deallocate descriptor 'desc1';
```

DESCRIBE INPUT statement

Use the DESCRIBE INPUT statement to return input parameter information before a prepared statement is executed.

Use this statement with ESQL/C.

Syntax

(explicit id unique_635_Connect_42_desi001) unique_635_Connect_42_desi001 **DESCRIBE INPUT** { *statement_var* | *statement_id* } { **USINGSQL DESCRIPTOR** { *'descriptor'* | *descriptor_var* } | **INTO** { **SQL DESCRIPTOR** { *'descriptor'* | *descriptor_var* } | [] *sqlda_pointer* } }

Element	Description	Restrictions	Syntax
descriptor	Name of a system-descriptor area	System-descriptor area must already be allocated	Quoted String on page 1202
descriptor_var	Host variable specifying a system-descriptor area	Must contain the name of an allocated system-descriptor area	Language-specific rules for names
sqlda_pointer	Pointer to an **sqlda** structure	Cannot begin with dollar ($) sign or colon (:). An **sqlda** structure is required if dynamic SQL is used.	See the **sqlda** structure in the *HCL® Informix® Enterprise Replication Guide*.
statement_id	Statement identifier for a prepared SQL statement	Must be defined in a previously executed PREPARE statement	PREPARE statement on page 623; PREPARE statement on page 623; Identifier on page 1231

Element	Description	Restrictions	Syntax
statement_var	Host variable that contains the value of *statement_id*	Variable and *statement_id* both must be declared	Language-specific rules for names

Usage

The DESCRIBE INPUT and the DESCRIBE OUTPUT statements can return information about a prepared statement to an SQL Descriptor Area (**sqlda**):

- For a SELECT, EXECUTE FUNCTION (or PROCEDURE), INSERT, or UPDATE statement, the DESCRIBE statement (with no INPUT keyword) returns the number, data types, and size of the returned values, and the name of the column or expression.
- For a SELECT, EXECUTE FUNCTION, EXECUTE PROCEDURE, DELETE, INSERT, or UPDATE statement, the DESCRIBE INPUT statement returns all the input parameters of a prepared statement.

Tip: Informix® versions earlier than 9.40 do not support the *INPUT* keyword. For compatibility with legacy applications, *DESCRIBE* without *INPUT* is supported. In new applications, you should use DESCRIBE INPUT statements to provide information about dynamic parameters in the WHERE clause, in subqueries, and in other syntactic contexts where the old form of DESCRIBE cannot provide information.

With this information, you can write code to allocate memory to hold retrieved values that you can display or process after they are fetched.

The **IFX_UPDDESC** environment variable does not need to be set before you can use DESCRIBE INPUT to obtain information about an UPDATE statement.

Describing the Statement Type

This statement takes a statement identifier from a PREPARE statement as input. After DESCRIBE INPUT executes, the SQLCODE field of the **sqlca** indicates the statement type (that is, the keyword with which the statement begins). If a prepared object contains more than one SQL statement, DESCRIBE INPUT returns the type of the first statement in the prepared text.

SQLCODE is set to zero to indicate a SELECT statement *without* an INTO TEMP clause. This situation is the most common. For any other SQL statement, SQLCODE is set to a positive integer. You can compare the number with the named constants that are defined in the **sqlstypes.h** header file.

The DESCRIBE and DESCRIBE INPUT statements use SQLCODE differently from other statements, under some circumstances returning a nonzero value after successful execution. You can revise standard error-checking routines to accommodate this behavior, if desired.

Checking for Existence of a WHERE Clause

If the DESCRIBE INPUT statement detects that a prepared object contains an UPDATE or DELETE statement without a WHERE clause, the database server sets the **sqlca.sqlwarn.sqlwarn4** variable to `W`.

When you do not specify a WHERE clause in either a DELETE or UPDATE statement, the database server performs the delete or update action on the entire table. Check the **sqlca.sqlwarn.sqlwarn4** variable after DESCRIBE INPUT executes to avoid unintended global changes to your table.

Describing a Statement with Dynamic Runtime Parameters

If the prepared statement specifies a set of parameters whose cardinality or data types must be supplied at runtime, you can describe these input values. If the prepared statement text includes one of the following statements, the DESCRIBE INPUT statement returns a description of each column or expression that is included in the list:

- EXECUTE FUNCTION (or EXECUTE PROCEDURE)
- INSERT or SELECT
- UPDATE or DELETE

The description includes the following information:

- The data type of the column, as defined in the table
- The length of the column, in bytes
- The name of the column or expression
- Information about *dynamic parameters* (parameters that are expressed as question (?) mark symbols within the prepared statement)

If the database server cannot infer the data type of an expression parameter, the DESCRIBE INPUT statement returns SQLUNKNOWN as the data type.

You can specify a destination for the returned information as a new or existing system-descriptor area, or as a pointer to an **sqlda** structure.

Using the SQL DESCRIPTOR Keywords

Specify INTO SQL DESCRIPTOR to create a new system-descriptor structure and store the description of a prepared statement list in that structure.

Use the USING SQL DESCRIPTOR clause to store the description of a prepared statement list in a previously allocated system-descriptor area. Executing the DESCRIBE INPUT . . . USING SQL DESCRIPTOR statement updates an existing system-descriptor area in the following ways:

- It allocates memory for the DATA field for each item descriptor, based on the TYPE and LENGTH information.
- It sets the COUNT field in the system-descriptor area to the number of values in the statement list. An error results if COUNT is greater than the number of item descriptors in the system-descriptor area.
- It sets the TYPE, LENGTH, NAME, SCALE, PRECISION, and NULLABLE fields in the system-descriptor area.

For columns of opaque data types, the DESCRIBE INPUT statement sets the EXTYPEID, EXTYPENAME, EXTYPELENGTH, EXTYPEOWNERLENGTH, and EXTYPEOWNERNAME fields of the item descriptor.

After a DESCRIBE INPUT statement executes, the SCALE and PRECISION fields contain the scale and precision of the column, respectively. If SCALE and PRECISION are set in the SET DESCRIPTOR statement, and TYPE is set to DECIMAL or MONEY, the LENGTH field is modified to adjust for the decimal scale and precision. If TYPE is not set to DECIMAL or MONEY, the values for SCALE and PRECISION are not set, and LENGTH is unaffected.

You must modify the system-descriptor-area information with the SET DESCRIPTOR statement to specify the address in memory that is to receive the described value. You can change the data type to another compatible type. This causes data conversion to take place when the data values are fetched.

You can also use the system-descriptor area in other statements that support a USING SQL DESCRIPTOR clause, such as EXECUTE, FETCH, OPEN, and PUT.

The following examples show the use of a system descriptor in a DESCRIBE statement. In the first example, the system descriptor is a quoted string; in the second example, it is an embedded variable name.

```
main()
{
. . .
EXEC SQL allocate descriptor 'desc1' with max 3;
EXEC SQL prepare curs1 FROM 'select * from tab';
EXEC SQL describe curs1 using sql descriptor 'desc1';
}
EXEC SQL describe curs1 using sql descriptor :desc1var;
```

A system-descriptor area conforms to the X/Open standards.

Using the INTO sqlda Pointer Clause

The INTO *sqlda_pointer* clause allocates memory for an **sqlda** structure and store its address in an **sqlda** pointer. The DESCRIBE INPUT statement fills in the allocated memory with descriptive information.

The DESCRIBE INPUT statement sets the **sqlda.sqld** field to the number of values in the statement list. The **sqlda** structure also contains an array of data descriptors (**sqlvar** structures), one for each value in the statement list. After a DESCRIBE statement is executed, the **sqlda.sqlvar** structure has the **sqltype**, **sqllen**, and **sqlname** fields set.

If the column has an opaque data type, DESCRIBE INPUT . . . INTO sets the **sqlxid**, **sqltypename**, **sqltypelen**, **sqlownerlen**, and **sqlownername** fields of the item descriptor.

The DESCRIBE INPUT statement allocates memory for an **sqlda** pointer once it is declared in a program. The application program, however, must designate the storage area of the **sqlda.sqlvar.sqldata** fields.

Describing a Collection Variable

The DESCRIBE INPUT statement can provide information about a collection variable if you use the INTO or USING SQL DESCRIPTOR clause.

You must execute the DESCRIBE INPUT statement *after* you open the Select or Insert cursor. Otherwise, DESCRIBE INPUT cannot get information about the collection variable because it is the OPEN . . . USING statement that specifies the name of the collection variable to use.

The next program fragment dynamically selects the elements of the **:a_set** collection variable into a system-descriptor area called **desc1**:

```
EXEC SQL BEGIN DECLARE SECTION;
      client collection a_set;
      int i, set_count;
      int element_type, element_value;
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate collection :a_set;
EXEC SQL allocate descriptor 'desc1';
EXEC SQL select set_col into :a_set from table1;
EXEC SQL prepare set_id from
      'select * from table(?)';


EXEC SQL declare set_curs cursor for set_id;
EXEC SQL open set_curs using :a_set;
EXEC SQL describe set_id using sql descriptor 'desc1';do
{
      EXEC SQL fetch set_curs using sql descriptor 'desc1';
      ...
      EXEC SQL get descriptor 'desc1' :set_count = count;
      for (i = 1; i <= set_count; i++)
      {
         EXEC SQL get descriptor 'desc1' value :i
            :element_type = TYPE;
         switch
         {
            case SQLINTEGER:
               EXEC SQL get descriptor 'desc1' value :i
                  :element_value = data;
            ...
         } /* end switch */
      } /* end for */
} while (SQLCODE == 0);

EXEC SQL close set_curs;
EXEC SQL free set_curs;
EXEC SQL free set_id;
EXEC SQL deallocate collection :a_set;
EXEC SQL deallocate descriptor 'desc1';
```

DISCONNECT statement

Use the DISCONNECT statement to terminate a connection between an application and a database server.

Syntax

(explicit id unique_650_Connect_42_discon001) unique_650_Connect_42_discon001 **DISCONNECT** { **CURRENT** | [] { **ALL** | **DEFAULT** } | { *'connection'* | *connection_var* } }

Element	Description	Restrictions	Syntax
connection	String that specifies a connection to terminate	Connection name that the CONNECT statement assigned	Quoted String on page 1202
connection_var	Host variable that holds the name of a connection	Must be a fixed-length character data type	Language specific

Usage

DISCONNECT terminates a connection to a database server. If a database is open, it closes before the connection drops. Even if you made a connection to a specific database only, the connection to the database server is terminated by DISCONNECT. If DISCONNECT does not terminate the current connection, the connection context of the current environment is not changed.

DISCONNECT is not valid as statement text in a PREPARE statement.

In ESQL/C, if you disconnect with *connection* or *connection_var*, DISCONNECT generates an error if the specified connection is not a current or dormant connection.

DEFAULT Option

DISCONNECT DEFAULT disconnects the default connection.

The *default connection* is one of the following connections:

- A connection established by the CONNECT TO DEFAULT statement
- An implicit default connection established by the DATABASE or CREATE DATABASE statement

You can use DISCONNECT to drop the default connection. If the DATABASE statement does not specify a database server, as in the following example, the default connection is made to the default database server:

```
EXEC SQL database 'stores_demo';
. . .
EXEC SQL disconnect default;
```

If the DATABASE statement specifies a database server, as the following example shows, the default connection is made to that database server:

```
EXEC SQL database 'stores_demo@mydbsrvr';
. . .
EXEC SQL disconnect default;
```

In either case, the DEFAULT option of DISCONNECT disconnects this default connection. For more information, see The DEFAULT Connection Specification on page 181.

Specifying the CURRENT Keyword

The DISCONNECT CURRENT statement terminates the current connection. For example, the DISCONNECT statement in the following program fragment terminates the current connection to the database server **mydbsrvr**:

```
CONNECT TO 'stores_demo@mydbsrvr';
. . .
DISCONNECT CURRENT;
```

When a Transaction is Active

DISCONNECT generates an error during a transaction. The transaction remains active, and the application must explicitly commit it or roll it back. If an application terminates without issuing DISCONNECT (because of a system failure or program error, for example), active transactions are rolled back.

In an ANSI-compliant database, however, if no error is encountered while you exit from in non-interactive mode without issuing the CLOSE DATABASE, COMMIT WORK, or DISCONNECT statement, the database server automatically commits any open transaction.

Disconnecting in a Thread-Safe Environment

If you issue the DISCONNECT statement in a thread-safe application, keep in mind that an active connection can be disconnected only from within the thread in which it is active. Therefore, one thread cannot disconnect the active connection of another thread. The DISCONNECT statement generates an error if such an attempt is made.

Once a connection becomes dormant, however, any other thread can disconnect it unless an ongoing transaction is associated with the dormant connection that was established with the WITH CONCURRENT TRANSACTION clause of CONNECT. If the dormant connection was not established with the WITH CONCURRENT TRANSACTION clause, DISCONNECT generates an error when it tries to disconnect the connection.

For an explanation of connections in a thread-safe application, see SET CONNECTION statement on page 779.

Specifying the ALL Option

Use the keyword ALL to terminate all connections established by the application up to that time. For example, the following DISCONNECT statement disconnects the current connection as well as all dormant connections:

```
DISCONNECT ALL;
```

In , the ALL keyword has the same effect on multithreaded applications that it has on single-threaded applications. Execution of the DISCONNECT ALL statement causes all connections in all threads to be terminated. However, the DISCONNECT ALL statement fails if any of the connections is in use or has an ongoing transaction associated with it. If either of these conditions is true, none of the connections is disconnected.

DROP ACCESS_METHOD statement

Use the DROP ACCESS_METHOD statement to remove a previously defined primary or secondary access method from the database.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
DROP ACCESS_METHOD [IF EXISTS] access_method RESTRICT
```

Element	Description	Restrictions	Syntax
access_method	Name of access method to drop	Must be registered in **sysams** system catalog table; cannot be a built-in access method	Identifier on page 1231
owner	Owner of the access method	Must own the access method	Owner name on page 1262

Usage

The RESTRICT keyword is required. You cannot drop an access method if virtual tables or indexes exist that use the access method. You must be the owner of the access method or have DBA privileges to drop an access method.

If a transaction is in progress, the database server waits to drop the access method until the transaction is committed or rolled back. No other users can execute the access method until the transaction has completed.

If you include the optional IF EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if no access method of the specified name is registered in the current database.

Example

For this example, suppose that an access method was created by this statement:

```
CREATE SECONDARY ACCESS_METHOD T_tree
(
  am_getnext = ttree_getnext,
  am_unique,
  am_cluster,
  am_sptype = 'S'
);
```

The following statement drops this access method:

```
DROP ACCESS_METHOD T_tree RESTRICT;
```

Details of existing access methods can be found in the **sysams** system catalog table with the following query:

```
SELECT am_name FROM informix.sysams;
```

DROP AGGREGATE statement

Use the DROP AGGREGATE statement to drop a user-defined aggregate that you created with the CREATE AGGREGATE statement.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

DROP AGGREGATE [IF EXISTS] [*owner* .] *aggregate*

Element	Description	Restrictions	Syntax
aggregate	Name of the user-defined aggregate to be dropped	Must have been previously created with the CREATE AGGREGATE statement	Identifier on page 1231
owner	Owner of the aggregate	Must own the aggregate	Owner name on page 1262

Usage

Dropping a user-defined aggregate does not drop the support functions that you defined for the aggregate in the CREATE AGGREGATE statement. The database server does not track dependency of SQL statements on user-defined aggregates that you use in the statements. For example, you can drop a user-defined aggregate that is used in an SPL routine.

The following example drops the user-defined aggregate named **my_avg**:

```
DROP AGGREGATE my_avg;
```

If you include the optional IF EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if no aggregate with the specified name is registered in the current database.

DROP CAST statement

Use the DROP CAST statement to remove an existing cast from the database.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
DROP CAST [IF EXISTS] ( source_type AS target_type )
```

Element	Description	Restrictions	Syntax
source_type	Data type that the cast accepts as input	Must exist	Identifier on page 1231; Data Type on page 985
target_type	Data type returned by the cast	Must exist	Identifier on page 1231; Data Type on page 985

Usage

You must be owner of the cast or have the DBA privilege to use DROP CAST. Dropping a cast removes its definition from the **syscasts** system catalog table, so the cast cannot be invoked explicitly or implicitly. Dropping a cast has no effect on the user-defined function associated with the cast. Use the DROP FUNCTION statement to remove the user-defined function from the database.

Warning: Do not drop built-in casts, which user **informix** owns. These are required for automatic conversions between built-in data types.

A cast defined on a given data type can also be used on any DISTINCT types created from that source type. If you drop the cast, you can no longer invoke it for the DISTINCT types, but dropping a cast that is defined for a DISTINCT type has no effect on casts for its source type. When you create a DISTINCT type, the database server automatically defines an explicit cast from the DISTINCT type to its source type and another explicit cast from the source type to the DISTINCT type. When you drop the DISTINCT type, the database server automatically drops these two casts.

If you include the optional IF EXISTS keywords, the database server takes no action (rather than sending an error to the application) if no cast between the two specified data types is registered in the current database.

Example

A cast (like this one in the **superstores_demo** database) can be dropped with the DROP CAST statement:

```
DROP CAST (decimal(5,5) AS percent);
```

Details of existing casts can be found in the **syscasts** system catalog table the following SQL:

```
SELECT routine_name, class, argument_type, result_type FROM Syscasts;
```

DROP DATABASE statement

Use the DROP DATABASE statement to delete an entire database, including all system catalog tables, objects, and data.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_659_Connect_42_drodb001) unique_659_Connect_42_drodb001 **DROP DATABASE [IF EXISTS]** <Database Name>[]

Usage

The DROP DATABASE statement is an extension to the ANSI/ISO standard, which does not provide syntax for the destruction of a database.

The following statement drops the **stores_demo** database:

```
DROP DATABASE IF EXISTS stores_demo;
```

You must have the DBA privilege or be user **informix** to run the DROP DATABASE statement successfully. Otherwise, the database server issues an error message and does not drop the database.

Restrictions on the DROP DATABASE statement

You cannot drop the current database or a database that is currently being used by another user. All the current users of the database must first execute the CLOSE DATABASE statement before DROP DATABASE can be successful.

You cannot drop a tenant database with the DROP DATABASE statement. You must drop a tenant database by running the admin() or task() SQL administration API function with the tenant drop argument.

The DROP DATABASE statement attempts to create an implicit connection to the database that you intend to drop. If a previous CONNECT statement has established an explicit connection to another database, and that connection is still your current connection, the DROP DATABASE statement fails with error -1811. In this case, you must first use the DISCONNECT statement to close the explicit connection before you can execute the DROP DATABASE statement.

If you include the optional IF EXISTS keywords, the database server takes no action (rather than returning an error to the application) if no database of the specified name is managed by the database server instance to which you are connected.

The DROP DATABASE statement cannot appear in a multistatement PREPARE statement, nor within an SPL routine.

In a DROP DATABASE operation, the database server acquires a lock on each table in the database and holds the locks until the entire operation is complete. Configure your database server with enough locks to accommodate this fact.

For example, if the database to be dropped has 2500 tables, but fewer than 2500 locks were configured for your database server, the DROP DATABASE statement fails. For more information on how to configure the number of locks available to the database server, see the discussion of the LOCKS configuration parameter in your *HCL® Informix® Administrator's Reference*.

In , use the DROP DATABASE statement with caution. does not prompt you to verify that you want to delete the entire database.

In ESQL/C, you can use an unqualified database name in a program or host variable, or you can specify the fully-qualified *database@server* format. For example, the following statement drops the **stores_demo** database of a database server called **gibson95**:

```
EXEC SQL DROP DATABASE stores_demo@gibson95;
```

If this statement executes successfully, the **gibson95** database server instance continues to exist, but the **stores_demo** database of that database server no longer exists. For more information, see Database Name on page 1224.

DROP FUNCTION statement

Use the DROP FUNCTION statement to remove a user-defined function from the database. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax



Element	Description	Restrictions	Syntax
function	Name of the user-defined function to be dropped	Must exist (that is, be registered) in the database. If the name does not uniquely identify a function, you must enter one or more appropriate values for *parameter_type*.	Identifier on page 1231

Element	Description	Restrictions	Syntax
parameter_type	Data type of the parameter	The data type (or list of data types) must be the same data types (and specified in the same order) as in the CREATE FUNCTION statement that registered the function	Data Type on page 985

Usage

Dropping a user-defined function removes the text and executable versions of the function from the database. (Make sure to keep a copy of the function text somewhere outside the database, in case you need to re-create a function after it is dropped.)

If you do not know whether a UDR is a function or a procedure, you can drop it by using the DROP ROUTINE statement.

To use the DROP FUNCTION statement, you must be the owner of the user-defined function (and hold the Resource privilege on the database) or have the DBA privilege. You must also hold the Usage privilege on the programming language in which the UDR is written. To drop an external user-defined function, see also Dropping an External Routine on page 483.

You cannot use the DROP ROUTINE, DROP FUNCTION, or DROP PROCEDURE statement to drop a protected routine. For more information about protected routines, see the description of the **sysprocedures** system catalog table in the *HCL® Informix® Guide to SQL: Reference*.

You cannot drop an SPL function from within the same SPL function.

Informix® can resolve a function by its *specific name*, if the function definition declared a specific name. If you use the specific name in this statement, you must also use the keyword SPECIFIC, as in the following example:

```
DROP SPECIFIC FUNCTION compare_point;
```

Otherwise, if the *function* name is not unique within the database, you must specify enough *parameter_type* information to disambiguate the name. If you use parameter data types to identify a user-defined function, they follow the function name, as in the following example:

```
DROP FUNCTION compare (int, int);
```

But the database server returns an error if it cannot resolve an ambiguous function name whose signature differs from that of another function only in an unnamed ROW-type parameter. (This error cannot be anticipated by the database server when the ambiguous *function* is defined.)

If you include the optional IF EXISTS keywords, the database server takes no action (rather than issue an error) if the database server cannot find in the current database any function that matches what your DROP FUNCTION statement specifies.

Determine whether a function exists

Before you attempt to drop a user-defined function, you can check for its existence in the database by querying the system catalog. In the following example, the SELECT statement retrieves from the **sysprodures** table any routines whose identifier is `MyFunction`:

```
SELECT * FROM sysprocedures WHERE procname = MyFunction;
```

If this query returns a single row, then a UDR called `MyFunction` is registered in the current database.

If this query returns no rows, you do not need to issue the DROP FUNCTION statement, but you might wish to verify that the WHERE clause specified the correct name, and that you are connected to the correct database.

If the query returns more than one row, then the routine name `MyFunction` is overloaded in the current database, and you need to examine the attributes of the `MyFunction` routines to determine which of them, if any, need to be unregistered by the DROP FUNCTION statement.

Examples

Most functions can be dropped using SQL statements similar to the following:

```
DROP FUNCTION best_month;
```

If you have more than one function with the same name, however, by using function overloading, the DROP FUNCTION statement must either specify the specific name of the function (if it has one), or the parameter list to uniquely identify it. For example, the **superstores_demo** database has two **last_contact** functions that were created with the following arguments:

```
CREATE FUNCTION last_contact(cust_name name_t) ...
```

and

```
CREATE FUNCTION last_contact(c_num INT) ...
```

To drop the second of these functions, use the following:

```
DROP FUNCTION last_contact(INT);
```

If the above functions had been created with the specific names **last_cname_contact** and **last_cnum_contac**t, then to drop the second of these, issue the following statement:

```
DROP SPECIFIC FUNCTION last_cnum_contact;
```

Details of existing functions can be found in the **sysprocedures** system catalog table using SQL queries like the following:

```
SELECT procname, specificname, paramtypes
         FROM sysprocedures ;
```

Dropping External Functions

A user-defined function (UDF) written in C language or in the Java™ language is called an *external function*. External functions must include the External Routine Reference clause that specifies a shared-object filename. By default, only users to whom the DBSA has granted the built-in EXTEND role can create or drop an external function. You must also hold the Usage privilege on the external programming language in which the UDF is written. See the section Granting the EXTEND Role on page 555 for additional information about the EXTEND role security feature. See the section Language-Level Privileges on page 549 for the syntax of the USAGE ON LANGUAGE clause for the C language or for the Java™ language.

To remove the executable version of a C language routine from shared memory, call the **IFX_UNLOAD_MODULE** function. To replace the executable version of a C routine with another routine, call the **IFX_REPLACE_MODULE** function. Both of these built-in functions are described in UDR Definition Routines.

DROP INDEX statement

Use the DROP INDEX statement to remove an index.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
DROP INDEX [IF EXISTS] [ owner . ] index [ONLINE]
```

Element	Description	Restrictions	Syntax
index	Name of the index to be dropped	Must exist in the database	Identifier on page 1231
owner	Name of index owner	Must own the index	Owner name on page 1262

Usage

In a typical online transaction processing (OLTP) environment, concurrent applications are connected to the database server to perform DML operations. For every query, the optimizer chooses a plan that is based on existing indexes, distribution statistics, and directives. After numerous OLTP transactions, however, the chosen plan might no longer be the best plan for query execution. In this case, dropping an inefficient index can sometimes improve performance.

You must be the owner of the *index* or have the DBA privilege to use the DROP INDEX statement. The following example drops the index **o_num_ix** that **joed** owns. The **stores_demo** database must be the current database:

```
DROP INDEX stores_demo:joed.o_num_ix;
```

You cannot use the DROP INDEX statement to drop a unique constraint, nor to drop an index that supports a constraint; you must use the ALTER TABLE . . . DROP CONSTRAINT statement to drop the constraint. When you drop the constraint, the database server automatically drops any index that exists solely to support that constraint. If you attempt to use DROP INDEX to drop an index that is shared by a unique constraint, the database server renames the specified index in the **sysindexes** system catalog table, declaring a new name in this format:

```
[space]<tabid>_<constraint_id>
```

Here *tabid* and *constraint_id* are from the **systables** and **sysconstraints** system catalog tables, respectively. The **sysconstraints.idxname** column is then updated to something like: `" 121_13"` (where quotation marks show the leading blank space). If this index is a unique index with only referential constraints sharing it, the index is downgraded to a duplicate index after it is renamed.

In some contexts, an alternative to the DROP INDEX statement is the SET Database Object Mode statement, which can disable a specified index without removing it from the system catalog. For more information about this SQL statement, which can also enable an index that is currently disabled, see SET Database Object Mode statement on page 784.

If you include the optional IF EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if no index of the specified name is registered in the current database.

Example

An index such as the one found in the *stores_demo* database can be dropped with:

```
DROP INDEX zip_ix;
```

If necessary, you can specify the index name as the fully qualified four-part object name (*database@instance:owner.indexname*), as in the following:

```
DROP INDEX stores_demo@prod:"informix".zip_ix ;
```

Details of existing functions can be found in the *sysprocedures* system catalog table, as in the following:

```
SELECT idxname FROM sysindices ;
```

The ONLINE keyword of DROP INDEX

For indexes that were not defined with the IN TABLE keyword option, the DBA can reduce the risk of nonexclusive access errors, and can increase the availability of the indexed table, by including the ONLINE keyword as the last specification of the DROP INDEX statement. The ONLINE keyword instructs the database server to drop the index while minimizing the duration of an exclusive lock, so that the index can be dropped while concurrent users are accessing the table.

By default, DROP INDEX attempts to place an exclusive lock on the indexed table to prevent all other users from accessing the table while the index is being dropped. The DROP INDEX statement fails if another user already has a lock on the table, or is currently accessing the table at the Dirty Read isolation level.

After you issue the DROP INDEX ONLINE statement, the query optimizer does not consider using the specified index in subsequent query plans or cost estimates, and the database server does not support any other DDL operations on the indexed table, until after the specified index has been dropped. Query operations that were initiated prior to the DROP INDEX ONLINE statement, however, can continue to access the index until the queries are completed.

When no other users are accessing the index, the database server drops the index, and the DROP INDEX ONLINE statement terminates execution.

By default, the DROP INDEX ONLINE statement does not wait indefinitely for locks to be released. If one or more concurrent sessions hold locks on the table, the statement might fail with error -216 or -113 unless you first issue the SET LOCK MODE TO WAIT statement to specify an indefinite wait. Otherwise, DROP INDEX ONLINE uses the waiting period for locks that the DEADLOCK_TIMEOUT configuration parameter specifies, or that a previous SET LOCK MODE statement specified. To avoid locking errors, execute SET LOCK MODE TO WAIT (with no specified limit) before you attempt to drop an index online.

You cannot use the CREATE INDEX statement to declare a new index that has the same identifier until after the specified index has been dropped. No more than one CREATE INDEX ONLINE or DROP INDEX ONLINE statement can concurrently reference indexes on the same table.

The indexed table in a DROP INDEX ONLINE statement can be permanent or temporary, logged or unlogged, and fragmented or non-fragmented. You cannot specify the ONLINE keyword, however, when you are dropping an index that has any of the following attributes:

- a functional index
- a clustered index
- a virtual index
- an R-tree index.

The following statement instructs the database server to drop online an index called **idx_01**:

```
DROP INDEX IF EXISTS idx_01 ONLINE;
```

For indexes defined with the IN TABLE keyword option, however, the indexed table remains locked for the duration of the DROP INDEX ONLINE operation. While the in-table index is being dropped, attempted access by other sessions to the locked table would fail with one or more of these errors:

```
107: ISAM error:  record is locked.
211: Cannot read system catalog (systables).
710: Table (table.tix) has been dropped, altered or renamed.
```

DROP OPCLASS statement

Use the DROP OPCLASS statement to remove an existing operator class from the database.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

DROP OPCLASS [IF EXISTS] [*owner* .] *opclass* **RESTRICT**

Element	Description	Restrictions	Syntax
opclass	Name of operator class to be dropped	Must have been created by a previous CREATE OPCLASS statement	Identifier on page 1231
owner	Name of opclass owner	Must own the operator class	Owner name on page 1262

Usage

You must be the owner of the operator class or have the DBA privilege to use the DROP OPCLASS statement.

If you include the optional IF EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if no operator class of the specified name is registered in the current database.

The RESTRICT keyword causes DROP OPCLASS to fail if the database contains indexes defined on the operator class that you plan to drop. Therefore, before you drop the operator class, you must use the DROP INDEX statement to drop any dependent indexes.

The following DROP OPCLASS statement drops an operator class called **abs_btree_ops**:

```
DROP OPCLASS abs_btree_ops RESTRICT
```

If you include the optional IF EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if no function of the specified name (or of the specified specific name, if you include the SPECIFIC keyword) is registered in the current database.

DROP PROCEDURE statement

Use the DROP PROCEDURE statement to drop a user-defined procedure from the database. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_668_Connect_42_drpr002) unique_668_Connect_42_drpr002(explicit id unique_668_Connect_42_drpr004) unique_668_Connect_42_drpr004

DROP { **PROCEDURE [IF EXISTS]** [*owner* .] { *procedure* | *function* } [(*parameter_type*)] | **SPECIFIC PROCEDURE [IF EXISTS]** <Specific Name> }

Element	Description	Restrictions	Syntax
function	Name of a procedure or SPL function to drop	Must exist (that is, be registered) in the database	Identifier on page 1231
owner	Name of UDR owner	Must own the procedure or SPL function	Owner name on page 1262
parameter_type	The data type of the parameter	The data type (or list of data types) must be the same types (and in the same order) as those specified when the procedure was created	Identifier on page 1231; Data Type on page 985
procedure	Name of user-defined procedure to drop	Must exist (that is, be registered) in the database	Database Object Name on page 1225

Usage

Dropping a user-defined procedure removes the text and executable version of the procedure from the database. You cannot drop an SPL procedure within the same SPL procedure.

You cannot use the DROP ROUTINE, DROP FUNCTION, or DROP PROCEDURE statement to drop a protected routine. For more information about protected routines, see the description of the **sysprocedures** system catalog table in the *HCL® Informix® Guide to SQL: Reference*.

To use the DROP PROCEDURE statement, you must be the owner of the procedure and also hold the Resource privilege on the database, or have the DBA privilege. You must also hold the Usage privilege on the programming language in which the UDR is written. To drop an external user-defined procedure, see also Dropping an External Procedure on page 480.

If the *function* or *procedure* name is not unique within the database, you must specify enough *parameter_type* information to disambiguate the name. If the database server cannot resolve an ambiguous UDR name whose signature differs from that of another UDR only in an unnamed ROW type parameter, an error is returned. (This error cannot be anticipated by the database server when the ambiguous *function* or *procedure* is defined.)

If you do not know whether a UDR is a user-defined procedure or a user-defined function, you can use the DROP ROUTINE statement. For more information, see DROP ROUTINE statement on page 481.

For compatibility with earlier Informix® versions, you can use this statement to drop an SPL function that CREATE PROCEDURE created. You can include parameter data types after the name of the procedure to identify the procedure:

```
DROP PROCEDURE compare(int, int);
```

If you use the specific name for the user-defined procedure, you must also use the keyword SPECIFIC, as in the following example:

```
DROP SPECIFIC PROCEDURE compare_point;
```

If you include the optional IF EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if no procedure of the specified name is registered in the current database.

Determine whether a procedure exists

Before you attempt to drop a user-defined procedure, you can check for its existence in the database by querying the system catalog. In the following example, the SELECT statement retrieves from the **sysprodures** table any routines whose identifier is `MyProcedure`:

```
SELECT * FROM sysprocedures WHERE procname = MyProcedure;
```

If this query returns a single row, then a UDR called `MyProcedure` is registered in the current database.

If this query returns no rows, you do not need to issue the DROP PROCEDURE statement, but you might wish to verify that the WHERE clause specified the correct name, and that you are connected to the correct database.

If the query returns more than one row, then the routine name `MyProcedure` is overloaded in the current database, and you need to examine the attributes of the `MyProcedure` routines to determine which of them, if any, need to be unregistered by the DROP PROCEDURE statement.

Dropping an External Procedure

A user-defined procedure (UDP) written in C language or in the Java™ language is called an *external routine*. External routines must include the External Routine Reference clause that specifies a shared-object filename. By default, only users to whom the DBSA has granted the built-in EXTEND role can create or drop an external routine. You must also hold the Usage privilege on the external programming language in which the UDP is written. See the section Granting the EXTEND Role on page 555 for additional information about the EXTEND role security feature. See the section unctions are described in Language-Level Privileges on page 549 for the syntax of the USAGE ON LANGUAGE clause for the C language or for the Java™ language.

To remove the executable version of a C language procedure from shared memory, call the **IFX_UNLOAD_MODULE** function. To replace the executable version of a C routine with another routine, call the **IFX_REPLACE_MODULE** function. Both of these built-in functions are described in UDR Definition Routines.

DROP ROLE statement

Use the DROP ROLE statement to remove a user-defined role from the database.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
DROP ROLE [IF EXISTS] { role | 'role' }
```

Element	Description	Restrictions	Syntax
role	Name of the role to be dropped	Must be registered in the local database. When a *role* name is enclosed in quotation marks, it is case sensitive.	Owner name on page 1262

Usage

Either the DBA or a user to whom the role was granted with the WITH GRANT OPTION keywords can issue the DROP ROLE statement. (Like a *user* name, a *role* is an authorization identifier, not a database object, so a *role* has no owner.)

If you include the optional IF EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if no role of the specified name is registered in the current database.

After you drop a role, no user can grant or enable the dropped role, and any user who had been assigned the role loses its privileges (such as table-level privileges or routine-level privileges) when the role is dropped, unless the same privileges were granted to PUBLIC or to the user individually. If the dropped role was the default role of a user, the default role for that user becomes NULL.

The following statement drops the role **engineer**:

```
DROP ROLE engineer;
```

You cannot use the DROP ROLE statement to drop a built-in role, such as the EXTEND or DBSECADM roles of Informix®.

DROP ROUTINE statement

Use the DROP ROUTINE statement to remove a user-defined routine (UDR) from the database.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
(explicit id unique_670_Connect_42_drro002b) unique_670_Connect_42_drro002b DROP { ROUTINE [IF EXISTS] [ owner . ] routine [ ( parameter_type ) ] | SPECIFIC ROUTINE [IF EXISTS] <Specific Name> }
```

Element	Description	Restrictions	Syntax
owner	Name of UDR owner	Must own the UDR	Owner name on page 1262
parameter_type	Data type of a parameter of *routine*	The data type (or list of data types) must be the same type (and specified in the same order) as in the UDR definition	Identifier on page 1231; Data Type on page 985
routine	Name of the UDR to drop	The UDR must exist (that is, be registered) in the database	Identifier on page 1231

Usage

Dropping a UDR removes the text and executable versions of the UDR from the database. If you do not know whether a UDR is a user-defined function or a user-defined procedure, this statement instructs the database server to drop the specified user-defined function or user-defined procedure.

To use the DROP ROUTINE statement, you must be the owner of the UDR (and also hold the Resource privilege on the database), or you must have the DBA privilege. You must also hold the Usage privilege on the programming language in which the UDR is written. To drop an external user-defined routine, see also Dropping an External Routine on page 483.

Restrictions

You cannot drop an SPL routine from within the same SPL routine.

You cannot use the DROP ROUTINE, DROP FUNCTION, or DROP PROCEDURE statement to drop a protected routine. For more information about protected routines, see the description of the **sysprocedures** system catalog table in the *HCL® Informix® Guide to SQL: Reference*.

To use the DROP ROUTINE statement to unregister a UDR, the type of UDR cannot be ambiguous. The name of the UDR that you specify must refer to either a user-defined function or a user-defined procedure. If either of the following conditions exist, the database server returns an error:

- The name (and parameters) that you specify apply to both a user-defined procedure and a user-defined function,
- The *specific* name that you specify applies to both a user-defined procedure and a user-defined function.

If the *routine* name is not unique within the database, you must specify enough *parameter_type* information to disambiguate the name. If the database server cannot resolve an ambiguous routine name whose signature differs from that of another routine only in an unnamed ROW type parameter, an error is returned. (This error cannot be anticipated by the database server when the ambiguous *routine* is defined.)

If you use parameter data types to identify a UDR, they follow the UDR name, as in the following example:

```
DROP ROUTINE compare(INT, INT);
```

If you use the specific name for the UDR, you must also include the keyword SPECIFIC, as in the following example:

```
DROP SPECIFIC ROUTINE compare_point;
```

If you include the optional IF EXISTS keywords, the database server takes no action (rather than issues an error) if the database server cannot find in the current database any routine that matches what your DROP ROUTINE statement specifies.

Determining Whether a Routine Exists

Before you attempt to drop a user-defined routine, you can check for its existence in the database by querying the system catalog. In the following example, the SELECT statement retrieves from the **sysprodures** table any routines whose identifier is **MyRoutine**:

```
SELECT * FROM sysprocedures WHERE procname = MyRoutine;
```

If this query returns a single row, then a UDR called **MyRoutine** is registered in the current database.

If this query returns no rows, you do not need to issue the DROP ROUTINE statement, but you might wish to verify that the WHERE clause specified the correct name, and that you are connected to the correct database.

If the query returns more than one row, then the routine name **MyRoutine** is overloaded in the current database, and you need to examine the attributes of the **MyRoutine** routines to determine which of them, if any, need to be unregistered by the DROP ROUTINE statement.

Dropping an External Routine

About this task

A user-defined routine (UDR) written in C language or in the Java™ language is called an *external routine*. External routines must include the External Routine Reference clause that specifies a shared-object filename. By default, only users to whom the DBSA has granted the built-in EXTEND role can create or drop an external routine. You must also hold the Usage privilege on the external programming language in which the UDR is written. See the section Granting the EXTEND Role on page 555 for additional information about the EXTEND security feature. See the section Language-Level Privileges on page 549 for the syntax of the USAGE ON LANGUAGE clause for the C language or for the Java™ language.

To remove the executable version of a C language procedure from shared memory, call the **IFX_UNLOAD_MODULE** function. To replace the executable version of a C routine with another routine, call the **IFX_REPLACE_MODULE** function. Both of these built-in functions are described in UDR Definition Routines.

DROP ROW TYPE statement

Use the DROP ROW TYPE statement to remove an existing named ROW data type from the database.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
DROP ROW TYPE [IF EXISTS] [ owner . ] row_type RESTRICT
```

Element	Description	Restrictions	Syntax
owner	Authorization identifier of the owner of *row_type*	Must own *row_type*	Owner name on page 1262
row_type	Name of an existing named ROW data type to be dropped	Must exist. See also the **Usage** section that follows.	Identifier on page 1231; Data Type on page 985

Usage

The DROP ROW TYPE statement removes the entry for the specified *row_type* from the **sysxtdtypes** system catalog table. You must be the owner of the specified named ROW data type or have the DBA privilege to execute the DROP ROW TYPE statement.

If you include the optional IF EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if no named ROW data type of the specified name is registered in the current database.

You cannot drop a named ROW data type if its name is in use in the database. The DROP ROW TYPE statement fails to destroy the specified ROW data type when any of the following conditions are true in the database:

- Any existing tables or columns are using the named ROW data type.
- The named ROW data type is a supertype in an inheritance hierarchy.
- A view is defined on a column of the named ROW data type.

Example of dropping a named ROW type

If the named ROW type **postal_t** is registered in the system catalog of the database, but none of the restrictions listed above apply, then the following statement destroys the ROW data type whose name is **postal_t**:

```
DROP ROW TYPE postal_t RESTRICT;
```

If this DROP ROW TYPE statement succeeds, the row in the system catalog where the named ROW type **postal_t** is registered is deleted from the **sysextdtypes** table.

To drop a named ROW-type column from the schema of a table without removing the corresponding ROW type database object from the system catalog, use the ALTER TABLE . . . DROP COLUMN statement.

The DROP ROW TYPE statement cannot drop unnamed ROW data types.

The RESTRICT Keyword

The RESTRICT keyword is required with the DROP ROW TYPE statement. RESTRICT causes DROP ROW TYPE to fail if dependencies on *row_type* exist.

The DROP ROW TYPE statement fails and returns an error message if any of the following conditions is true:

- The named ROW data type is used for an existing table or column.

  Check the **systables** and **syscolumns** system catalog tables to find out whether any tables or data types use the named ROW data type.

- The named ROW data type is the supertype in an inheritance hierarchy.

  Look in the **sysinherits** system catalog table to see which named ROW data types have child types.

The following statement drops the named ROW data type **employee_t**:

```
DROP ROW TYPE employee_t RESTRICT
```

DROP SECURITY statement

Use the DROP SECURITY statement to remove an existing security object from the current database. The object can be a security policy, security label, or a security label component.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

DROP SECURITY { { **LABEL [IF EXISTS]** *policy* . *label* | **LABEL COMPONENT [IF EXISTS]** *component* } **RESTRICT** | **POLICY [IF EXISTS]** *policy* { **RESTRICT** | **CASCADE** } }

Element	Description	Restrictions	Syntax
component	Security label component to drop	Must exist in the database	Identifier on page 1231
label	Security label to drop	Must exist in the database as a label of the specified *policy*	Identifier on page 1231
policy	Security policy to drop	Must exist in the database	Identifier on page 1231

Usage

Only DBSECADM can issue this statement. When the DROP SECURITY statement executes successfully, the database server deletes any rows that reference the name or the numeric identifier of the specified object from the tables of the system catalog, including these tables:

- **sysecpolicies** for security policies
- **sysseclabels** for security labels
- **sysseclabelcomponents** for security label components.

The keyword or keywords that follow the SECURITY keyword identify the type of security object that is being dropped.

- SECURITY POLICY *policy* specifies a security policy
- SECURITY LABEL *policy*.*label* specifies a security label
- SECURITY LABEL COMPONENT *component* specifies a security label component.

There is no SQL statement that selectively drops some elements of a security label component without destroying the entire component object. To remove only a subset of the elements of a security label component from the database, DBSECADM can use the DROP SECURITY LABEL COMPONENT statement to drop the component, and then redefine the dropped component, using the CREATE SECURITY LABEL COMPONENT statement, but without including any elements that are no longer needed. (An alternative is to drop all the security labels that include the deprecated elements, and then use the CREATE SECURITY LABEL statement to redefine new labels with the same components as the dropped labels, but without those elements. In this case, the deprecated elements persist in the database, but no security label uses them as values for their component.)

If you include the optional IF EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if no security object of the specified security object type and of the specified name is registered in the current database.

Examples

The following statement instructs the database server to drop the security label **witty**:

```
DROP SECURITY LABEL witty;
```

The statement fails if any column is protected by the **witty** label, or if any user holds this label.

The next example instructs the database server to drop the security label component **adhesive** from the database:

```
DROP SECURITY LABEL COMPONENT adhesive;
```

The statement fails if any security policy depends on the **adhesive** security label component.

The following example instructs the database server to drop the **best** security policy in CASCADE mode:

```
DROP SECURITY POLICY best CASCADE;
```

This statement fails if that policy is currently protecting any table. If this statement succeeds, however, it has the following additional effects because of the CASCADE specification:

- All security labels associated with the **best** security policy are also dropped.
- All exemptions from the **best** security policy are revoked.
- All security labels that were dropped because the **best** security policy was dropped are revoked from all users who hold those labels.

Dropping security objects in RESTRICT mode or in CASCADE mode

By default, the RESTRICT keyword is in effect when any security object is dropped. Only a security *policy* can be dropped in CASCADE mode. DBSECADM cannot drop a security policy in RESTRICT mode if any of the following conditions are true:

- A table is protected by that security policy
- A security label depends on that security policy
- A user has been granted an exemption from a rule of that security policy.

A security policy cannot be dropped in CASCADE mode if the policy is protecting any table. When a security policy is successfully dropped in CASCADE mode, the following security objects are also dropped or revoked:

- All the security labels that are associated with the dropped security policy
- All the security labels that were dropped are also revoked from the users who hold those labels
- All the exemptions from the dropped security policy are revoked.

A security label cannot be dropped in RESTRICT mode, which is the only supported mode for dropping labels, if any of the following conditions are true:

- A column is protected by that security label
- A user holds that security label.

A security label component cannot be dropped in RESTRICT mode, which is the only supported mode for dropping components, if any security policy depends on that security label component.

DROP SEQUENCE statement

Use the DROP SEQUENCE statement to remove a sequence object from the database.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
DROP SEQUENCE [IF EXISTS] [ owner . ] sequence
```

Element	Description	Restrictions	Syntax
owner	Name of sequence owner	Must own the sequence object	Owner name on page 1262
sequence	Name of a sequence	Must exist in the current database	Identifier on page 1231

Usage

This statement removes the *sequence* entry from the **syssequences** system catalog table. To drop a sequence, you must be its owner or have the DBA privilege on the database. In an ANSI-compliant database, you must qualify the name of the sequence with the name of its owner (*owner.sequence*) if you are not the owner.

If you include the optional IF EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if no sequence object of the specified name is registered in the current database.

If you drop a sequence, any synonyms for the name of the sequence are also dropped automatically by the database server.

You cannot use a synonym to specify the identifier of the *sequence* in the DROP SEQUENCE statement.

Examples

Suppose you had a sequence created with something code similar to the following:

```
CREATE SEQUENCE Invoice_Numbers
  START 10000 INCREMENT 1 NOCYCLE ;
```

Such a sequence can be dropped using the following:

```
DROP SEQUENCE Invoice_Numbers;
```

Details of existing sequences can be found by joining the *syssequences* and *systables* system catalog tables as in the following:

```
SELECT t.tabname SeqName
  FROM Syssequences s, Systables t
  WHERE t.tabid = s.tabid ;
```

DROP SYNONYM statement

Use the DROP SYNONYM statement to unregister an existing synonym.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

DROP SYNONYM [IF EXISTS] [*owner* .] *synonym*

Element	Description	Restrictions	Syntax
owner	Owner of *synonym*	Must own *synonym*	Owner name on page 1262
synonym	Synonym to be dropped	The synonym must exist in the current database.	Identifier on page 1231

Usage

This removes the entries for the *synonym* from the **systables** and **syssyntable** system catalog tables. You must own the *synonym* or have the DBA privilege to execute the DROP SYNONYM statement. Dropping a synonym has no effect on the table, view, or sequence object to which the synonym points.

If you include the optional IF EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if no synonym of the specified name is registered in the current database.

The following statement drops the synonym **nj_cust**, that user **cathyg** owns:

```
DROP SYNONYM cathyg.nj_cust;
```

DROP SYNONYM is not the only DDL operation that can unregister a synonym. If a table, view, or sequence is dropped, any synonyms that exist in the same database and that refer to that table, view, or sequence object are also dropped.

If a synonym in the current database refers to a dropped table or view in another database, however, that synonym remains registered in the system catalog until you explicitly drop it by using the DROP SYNONYM statement. You can create in the same database another table or view, and declare as its identifier the name of the dropped table or view. (If that is not the name of any table object in the current database, you can instead create a table, view, or sequence object in the current database, and declare as its name the identifier of the table or view that was dropped in the other database.) In either case, the old synonym now refers to the new table object. For a more complete discussion of synonym chaining, see the topic Chaining Synonyms on page 309 in the CREATE SYNONYM statement description.

DROP TABLE statement

Use the DROP TABLE statement to remove a table with its associated indexes and data. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
DROP TABLE [IF EXISTS] [ owner . ] { table | synonym } { CASCADE | [RESTRICT] }
```

Element	Description	Restrictions	Syntax
owner	Name of table owner	Must own the table	Owner name on page 1262
synonym	Local synonym for a table that is to be dropped	The synonym and its table must exist, and **USETABLENAME** must not be set to 1	Identifier on page 1231
table	Name of a table to drop	Must be registered in the **systables** system catalog table of the local database	Identifier on page 1231

Usage

You must be the owner of the table or have the DBA privilege to use the DROP TABLE statement.

If you include the optional IF EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if no table of the specified name is registered in the current database.

If DROP TABLE statement is used with a synonym name, both the synonym and the table identified by the synonym are dropped along with any other synonyms that reference the same table.

You can prevent users from specifying a synonym in DROP table statements by setting the USETABLENAME environment variable. If USETABLENAME is set, an error results if any user attempts to specify DROP TABLE *synonym*.

You cannot drop a system catalog table.

If you issue a DROP TABLE statement, does not prompt you to verify that you want to delete an entire table.

Effects of the DROP TABLE Statement

Use the DROP TABLE statement with caution. When you remove a table, you also delete the data stored in it, the indexes or constraints on the columns (including all the referential constraints placed on its columns), any local synonyms assigned to

it, any triggers created on it, and any access privileges granted on the table. You also drop all views based on the table and any violations and diagnostics tables associated with the table.

DROP TABLE does not remove any synonyms for the table that were created in an external database. To remove external synonyms for the dropped table, you must do so explicitly with the DROP SYNONYM statement.

If DROP TABLE statement is used with a synonym name, both the synonym and the table identified by the synonym are dropped along with any other synonyms that reference the same table.

You can prevent users from specifying a synonym in DROP TABLE statements by setting the **USETABLENAME** environment variable. If **USETABLENAME** is set, an error results if any user attempts to specify DROP TABLE *synonym*.

Specifying CASCADE Mode

The CASCADE keyword in DROP TABLE removes related database objects, including referential constraints built on the table, views defined on the table, and any violations and diagnostics tables associated with the table.

If the table is the supertable in an inheritance hierarchy, CASCADE drops all of the subtables as well as the supertable.

The CASCADE mode is the default mode of the DROP TABLE statement. You can also specify this mode explicitly with the CASCADE keyword.

Specifying RESTRICT Mode

The RESTRICT keyword can control the drop operation for supertables, for tables that have referential constraints and views defined on them, or for tables that have violations and diagnostics tables associated them. Using the RESTRICT option causes the drop operation to fail and an error message to be returned if any of the following conditions are true:

- Existing referential constraints reference *table.*
- Existing views are defined on *table.*
- Any violations tables or diagnostics tables are associated with *table*.
- The *table* is the supertable in an inheritance hierarchy.

Dropping a Table That Contains Opaque Data Types

Some opaque data types require special processing when they are deleted. For example, if an opaque type contains spatial or multi-representational data, it might provide a choice of how to store the data: inside the internal structure or, for large objects, in a smart large object.

The database server removes opaque types by calling a user-defined support function called **destroy( )**. When you execute the DROP TABLE statement on a table whose rows contain an opaque type, the database server automatically invokes the **destroy( )** function for the type. The **destroy( )** function can perform certain operations on columns of the opaque data type before the table is dropped. For more information about the **destroy( )** support function, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

Tables That Cannot Be Dropped

Restrictions exist on the types of tables that you can drop.

- You cannot drop any system catalog table.
- You cannot drop a table that is not in the current database.
- You cannot drop a violations table or a diagnostics table.

  Before you can drop such a table, you must first issue a STOP VIOLATIONS TABLE statement on the base table with which the violations and diagnostics tables are associated.

The following example removes two tables in the current database. Both are owned by **joed**, the current user. Neither table has an associated violations or diagnostics table, nor a referential constraint or view defined on it.

```
DROP TABLE customer;
DROP TABLE stores_demo@accntg:joed.state;
```

DROP TRIGGER statement

Use the DROP TRIGGER statement to remove a trigger definition from the database.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
DROP TRIGGER [IF EXISTS] [ owner . ] trigger
```

Element	Description	Restrictions	Syntax
owner	Name of the owner of the trigger	Must own the trigger	Owner name on page 1262
trigger	Name of the trigger to drop	The trigger must exist in the local database	Identifier on page 1231

Usage

You must be the owner of the trigger or have the DBA privilege to drop the trigger. Dropping a trigger removes the text of the trigger definition and the executable trigger from the database. The row describing the specified trigger is deleted from the **systriggers** system catalog table.

If you include the optional IF EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if no trigger of the specified name is registered in the current database.

Dropping an INSTEAD OF trigger on a complex view (a view with columns from more than one table) revokes any privileges on the view that the owner of the trigger received automatically when creating the trigger, and also revokes any privileges that the owner of the trigger granted to other users. (Dropping a trigger on a simple view does not revoke any privileges.)

The following statement drops the **items_pct** trigger:

```
DROP TRIGGER items_pct;
```

If a DROP TRIGGER statement appears inside an SPL routine that is called by a data manipulation (DML) statement, the database server returns an error.

When multiple triggers are defined on the same table or view for the same triggering event, the order in which the triggers execute is not guaranteed. If you have a preferred sequence of execution, but the triggers are executing in some other sequence, you might wish to drop all of the triggers except the one that you want to run first, and then re-create the other triggers in the relative order in which you want them to execute, so that they are listed in the system catalog in the intended order of execution.

DROP TRUSTED CONTEXT statement

Use the DROP TRUSTED CONTEXT statement to destroy a trusted-context object.

This statement is the Informix® extension to the ANSI/ISO standard for SQL. You must hold the database security administrator (DBSECADM) role to drop a trusted context.

Syntax

```
DROP TRUSTED CONTEXT context
```

Element	Description	Restrictions	Syntax
context	Trusted-context object to destroy.	Must exist on the current Informix® database server instance.	Identifier on page 1231

Usage

When the DROP TRUSTED CONTEXT statement executes successfully, the specified trusted context object is destroyed. All references to the *context* will be deleted from these tables in the **sysuser** database of the Informix® database server instance:

- **systrustedcontext**
- **systcxattributes**
- **systcxusers**.

If you drop the trusted context while trusted connections for this context are active, those connections remain trusted until they terminate, or until the next reuse attempt. If an attempt is made to switch the user on these trusted connections, however, an error is returned.

The following example of a DROP TRUSTED CONTEXT statement destroys the cntx1 trusted-context object:

```
DROP TRUSTED CONTEXT cntx1;
```

This example fails if cntx1 is not the name of a trusted-context object of the current database server instance.

DROP TYPE statement

Use the DROP TYPE statement to remove a user-defined distinct or opaque data type from the database. (You cannot use this statement to remove a built-in data type or a ROW data type.)

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
DROP TYPE [IF EXISTS] [ owner . ] data_type RESTRICT
```

Element	Description	Restrictions	Syntax
data_type	Name of distinct or opaque data type to be removed	Must be an existing user-defined DISTINCT or OPAQUE type in the local database. Cannot be a built-in data type.	Identifier on page 1231
owner	Authorization identifier of the *data type* owner	Must own *data type*	Owner name on page 1262

Usage

To drop a distinct or opaque data type with the DROP TYPE statement, you must be the owner of the data type or have the DBA privilege. When you use this statement, you remove the data type definition from the database (in the **sysxtdtypes** system catalog table). In general, this statement does not remove any definitions of casts or of support functions associated with that data type.

Important: When you drop a distinct type, the database server automatically drops the two explicit casts between the distinct type and the type on which it is based.

If you include the optional IF EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if no user-defined distinct or opaque data type of the specified name is registered in the current database.

The DROP TYPE statement fails with an error if you attempt to drop a built-in data type, such as the built-in opaque BOOLEAN or LVARCHAR type, or the built-in distinct IDSSECURITYLABEL type.

Do not confuse the DROP TYPE statement with the DROP ROW TYPE statement on page 483, which can only destroy named ROW types, including ROW types within data type hierarchies.

You cannot drop a distinct or opaque type if the database contains any casts, columns, or user-defined functions whose definitions reference the data type.

The following statement destroys the **new_type** data type:

```
DROP TYPE new_type RESTRICT;
```

DROP USER statement (UNIX™, Linux™)

Use the DROP USER statement to remove an internal user.

This statement is an extension to the ANSI/ISO standard for the SQL language.

Syntax

```
DROP USER user
```

Element	Description	Restrictions	Syntax
user	Authorization identifier of a specific user that you are dropping.	Must be an existing authorization identifier	Owner name on page 1262

Usage

Only a DBSA can run the DROP USER statement. With a non-root installation, the user who installs the server is the equivalent of the DBSA, unless the user delegates DBSA privileges to a different user.

It is recommended that you do not run the DROP USER statement while the specified user is active on a connection.

Execution of the DROP USER statement can be audited with the DRUR audit code.

Example

Example

The following statement drops the user **bill**:

```
DROP USER bill;
```

DROP VIEW statement

Use the DROP VIEW statement to remove a view from the database.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
DROP VIEW [IF EXISTS] [ owner . ] { view | synonym } { CASCADE | [RESTRICT] }
```

Element	Description	Restrictions	Syntax
owner	Name of view owner	Must own the view	Owner name on page 1262
synonym	Synonym for a view that this statement drops	The *synonym* and the view to which it points must exist in the local database	Identifier on page 1231
view	Name of a view to drop	Must exist in **systables**	Identifier on page 1231

Usage

To drop a view, you must be the owner or have the DBA privilege.

When you drop a view, you also drop any other views and INSTEAD OF triggers whose definitions depend on that view. (You can also specify this default behavior explicitly with the CASCADE keyword.)

If you include the optional IF EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if no view of the specified name is registered in the current database.

When you include the RESTRICT keyword in the DROP VIEW statement, the drop operation fails if any other existing views are defined on *view*; otherwise, these dependent views would be unregistered by the DROP VIEW operation.

You can query the **sysdepend** system catalog table to determine which views, if any, depend on another view.

The following statement drops the view that is named **cust1**:

```
DROP VIEW cust1
```

DROP XADATASOURCE statement

Use the DROP XADATASOURCE statement to drop a previously defined XA-compliant data source from the system catalog of the database.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
DROP XADATASOURCE [IF EXISTS] xa_source RESTRICT
```

Element	Description	Restrictions	Syntax
xa_source	The XA-compliant data source to drop	Must be present in the **sysxadatasources** system catalog table	Identifier on page 1231

Usage

The RESTRICT keyword is required. You must be the owner of the XA data source or hold DBA privileges to drop an access method.

The DROP XADATASOURCE statement is not supported on secondary servers within a high-availability cluster.

If you include the optional IF EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if no XA data source of the specified name is registered in the current database.

The following statement drops the XA data source instance called **NewYork** that is owned by user **informix**.

```
DROP XADATASOURCE informix.NewYork RESTRICT;
```

You cannot drop an access method if it is being used in a transaction that is currently open. If an XA data source has been registered with a transaction that is not complete, you can drop the data source only after the database is closed or the session ends.

DROP XADATASOURCE TYPE statement

Use the DROP XADATASOURCE TYPE statement to drop a previously defined XA-compliant data source type from the database.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
DROP XADATASOURCE TYPE [IF EXISTS] xa_type RESTRICT
```

Element	Description	Restrictions	Syntax
xa_type	Name of the XA data source type to be dropped	Must be present in the **sysxasourcetypes** system catalog table	Identifier on page 1231

Usage

The RESTRICT keyword is required. You cannot unregister an XA data source type if virtual tables or indexes exist that use the data source. You must be user **informix** or have DBA privileges to drop an XA data source type.

The DROP XADATASOURCE TYPE statement is not supported on secondary servers within a high-availability cluster.

The following statement drops an XA data source type called **MQSeries®** owned by user **informix**:

```
DROP XADATASOURCE TYPE informix.MQSeries RESTRICT;
```

You cannot drop an XA data source type until after all the XA data source instances that use that data source type have been dropped.

If you include the optional IF EXISTS keywords, the database server takes no action (rather than sending an exception to the application) if no XA data source type of the specified name is registered in the current database.

EXECUTE statement

Use the EXECUTE statement to run a previously prepared statement or a multiple-statement prepared object.

Use this statement with .

Syntax

```
(explicit id unique_690_Connect_42_ex001) unique_690_Connect_42_ex001(explicit id unique_690_Connect_42_ex002) unique_690_Connect_42_ex002

EXECUTE { stmt_id | stmt_id_var } [ <INTO Clause> ] [ <USING Clause> ]
```

Element	Description	Restrictions	Syntax
stmt_id	Identifier of a prepared SQL statement	Must have been declared in a previous PREPARE statement	Identifier on page 1231
stmt_id_var	Host variable containing the identifier of a prepared statement	Must exist and must contain a statement identifier that a previous PREPARE statement declared, and must be of a character data type	PREPARE statement on page 623

Usage

The EXECUTE statement passes a prepared SQL statement to the database server for execution. The following example shows an EXECUTE statement within the program:

```
EXEC SQL PREPARE del_1 FROM
   'DELETE FROM customer WHERE customer_num = 119';
EXEC SQL EXECUTE del_1;
```

Once prepared, an SQL statement can be executed as often as needed.

After you release the database server resources (using a FREE statement), you cannot use the statement identifier with a DECLARE cursor or with the EXECUTE statement until you prepare the statement again.

If the statement contained question mark (?) placeholders, use the USING clause to provide specific values for them before execution. For more information, see the #unique_692_Connect_42_sii-02exec-27987.

You can execute any prepared statement except those in the following list:

- A prepared SELECT statement that returns more than one row

  When you use a prepared SELECT statement to return multiple rows of data, you must use a cursor to retrieve the data rows. As an alternative, you can EXECUTE a prepared SELECT INTO TEMP statement to achieve the same result.

  For more information on cursors, see DECLARE statement on page 433.
- A prepared EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement for an SPL function that returns more than one row

  When you prepare an EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement to invoke an SPL function that returns multiple rows, you must use a cursor to retrieve the data rows.

  For more information on how to execute a SELECT or an EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement, see PREPARE statement on page 623.

If you create or drop a trigger after you prepare a triggering INSERT, DELETE, or UPDATE statement, the prepared statement returns an error when you execute it.

Scope of Statement Identifiers

A program can consist of one or more source-code files. By default, the scope of reference of a statement identifier is global to the program. A statement identifier created in one file can be referenced from another file.

In a multiple-file program, if you want to limit the scope of reference of a statement identifier to the file in which it is executed, you can preprocess all the files with the **-local** command-line option.

INTO Clause

Use the INTO clause to save the returned values of these SQL statements:

- A prepared singleton SELECT statement that returns only one row of column values for the columns in the select list
- A prepared EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement for an SPL function that returns only one set of values

The INTO clause of the EXECUTE statement has the following syntax:

(explicit id unique_691_Connect_42_uscl002) unique_691_Connect_42_uscl002

INTO Clause

" **INTO** { *output_var* [{ [] : | **INDICATOR** } *indicator_var*] | **SQL DESCRIPTOR** { *descriptor_var* | *'descriptor'* } | **DESCRIPTOR***sqlda_pointer* } "

Element	Description	Restrictions	Syntax
descriptor	Quoted string that identifies a system-descriptor area	Must already be allocated. Use single (') quotation marks	Quoted String on page 1202
descriptor_var	Host variable that identifies a system-descriptor area	System-descriptor area must already be allocated	Language specific
indicator_var	Host variable that receives a return code if corresponding *parameter_var* is NULL value, or if truncation occurs	Cannot be DATETIME or INTERVAL data type	Language specific
output_var	Host variable whose contents replace a question-mark (?) placeholder in a prepared statement	Must be a character data type	Language specific
sqlda_pointer	Pointer to an **sqlda** structure that defines data type and memory location of values to replace a question-mark (?) placeholder in a prepared object	Cannot begin with a dollar sign ($) or a colon (**:**) symbol. An **sqlda** structure is required with dynamic SQL	DESCRIBE INPUT statement on page 463

This closely resembles the syntax of the #unique_692_Connect_42_sii-02exec-27987.

The INTO clause provides a concise and efficient alternative to more complicated and lengthy syntax. In addition, by placing values into variables that can be displayed, the INTO clause simplifies and enhances your ability to retrieve and display data values. For example, if you use the INTO clause, you do not need to use a cursor to retrieve values from a table.

You can store the returned values in output variables, in output SQL descriptors, or in output **sqlda** pointers.

Restrictions with the INTO Clause

If you execute a prepared SELECT statement that returns more than one row, or execute a prepared EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement for an SPL function that returns more than one group of return values, you receive an error message. In addition, if you prepare and declare a statement and then attempt to execute that statement, you receive an error message.

You cannot select a NULL value from a table column and place that value into an output variable. If you know in advance that a table column contains a NULL value, after you select the data, check the indicator variable that is associated with the column to determine if the value is NULL.

To use the INTO clause with the EXECUTE statement:

1. Declare the output variables that the EXECUTE statement uses.
2. Use PREPARE to prepare your SELECT statement or to prepare your EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement.
3. Use the EXECUTE statement, with the INTO clause, to execute your SELECT statement or to execute your EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement.

Replacing Placeholders with Parameters

You can specify any of the following items to replace the question-mark placeholders in a prepared statement before you execute it:

- A host variable name (if the number and data type of the parameters are known at compile time)
- A system descriptor that identifies a system descriptor area
- A descriptor that is a pointer to an **sqlda** structure

Sections that follow describe each of these options for specifying parameters.

Saving Values In Host or Program Variables

If you know the number of return values to be supplied at runtime and their data types, you can define the values that the SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement returns as host variables in your program. Use these host variables with the INTO keyword, followed by the names of the variables. These variables are matched with the return values in a one-to-one correspondence, from left to right.

You must supply one variable name for each value that the SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) returns. The data type of each variable must be compatible with the corresponding returned value from the prepared statement.

Saving Values in a System-Descriptor Area

If you do not know the number of return values to be supplied at runtime or their data types, you can associate output values with a system-descriptor area. A system-descriptor area describes the data type and memory location of one or more values.

A system-descriptor area conforms to the X/Open standards.

To specify a system-descriptor area as the location of output values, use the INTO SQL DESCRIPTOR clause of the EXECUTE statement. Each time that the EXECUTE statement is run, the values that the system-descriptor area describes are stored in the system-descriptor area.

The following example shows how to use the system-descriptor area to execute prepared statements in :

```
EXEC SQL allocate descriptor 'desc1';
...
sprintf(sel_stmt, "%s %s %s",
   "select fname, lname from customer",
   "where customer_num =",
   cust_num);
EXEC SQL prepare sel1 from :sel_stmt;
EXEC SQL execute sel1 into sql descriptor 'desc1';
```

The COUNT field corresponds to the number of values that the prepared statement returns. The value of COUNT must be less than or equal to the value of the occurrences that were specified when the system-descriptor area was allocated with the ALLOCATE DESCRIPTOR statement.

You can obtain the value of a field with the GET DESCRIPTOR statement and set the value with the SET DESCRIPTOR statement.

For more information, refer to the discussion of the system-descriptor area in the *HCL® Informix® Enterprise Replication Guide*.

Saving Values in an sqlda Structure (ESQL/C)

If you do not know the number of output values to be returned at runtime or their data types, you can associate output values from an **sqlda** structure. An **sqlda** structure lists the data type and memory location of one or more return values. To specify an **sqlda** structure as the location of return values, use the INTO DESCRIPTOR clause of the EXECUTE statement. Each time the EXECUTE statement is run, the database server places the returns values that the **sqlda** structure describes into the **sqlda** structure.

The next example uses an **sqlda** structure to execute a prepared statement:

```
struct sqlda *pointer2;
...
sprintf(sel_stmt, "%s %s %s",
   "select fname, lname from customer",
   "where customer_num =",
   cust_num);
EXEC SQL prepare sel1 from :sel_stmt;
EXEC SQL describe sel1 into pointer2;
EXEC SQL execute sel1 into descriptor pointer2;
```

The **sqlda.sqld** value specifies the number of output values that are described in occurrences of **sqlvar**. This number must correspond to the number of values that the SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement returns.

For more information, refer to the **sqlda** discussion in the *HCL® Informix® Enterprise Replication Guide*.

This example uses the INTO clause with an EXECUTE statement in :

```
EXEC SQL prepare sel1 from 'select fname, lname from customer
    where customer_num =123';
EXEC SQL execute sel1 into :fname, :lname using :cust_num;
```

The next example uses the INTO clause to return multiple rows of data:

```
EXEC SQL BEGIN DECLARE SECTION;
int customer_num =100;
char fname[25];
EXEC SQL END DECLARE SECTION;

EXEC SQL prepare sel1 from 'select fname from customer
    where customer_num=?';
for ( ;customer_num < 200; customer_num++)
    {
    EXEC SQL execute sel1 into :fname using customer_num;
    printf("Customer number is %d\n", customer_num);
    printf("Customer first name is %s\n\n", fname);
    }
```

Supplying Parameters Through Host or Program Variables

You pass parameters to the database server by opening the cursor with the USING keyword, followed by the names of the variables. These variables are matched with prepared statement question-mark (?) placeholders in a one-to-one correspondence, from left to right. You must supply one storage- parameter variable for each placeholder. The data type of each variable must be compatible with the corresponding value that the prepared statement requires.

The following example executes the prepared UPDATE statement in :

```
stcopy ("update orders set order_date = ?
   where po_num = ?", stm1);
EXEC SQL prepare statement_1 from :stm1;
EXEC SQL execute statement_1 using :order_date, :po_num;
```

Supplying Parameters Through a System Descriptor

You can create a system-descriptor area that describes the data type and memory location of one or more values and then specify the descriptor in the USING SQL DESCRIPTOR clause of the EXECUTE statement.

Each time that the EXECUTE statement is run, the values that the system-descriptor area describes are used to replace question-mark (?) placeholders in the PREPARE statement.

The COUNT field corresponds to the number of dynamic parameters in the prepared statement. The value of COUNT must be less than or equal to the number of item descriptors that were specified when the system-descriptor area was allocated with the ALLOCATE DESCRIPTOR statement.

The following example shows how to use system descriptors to execute a prepared statement in :

```
EXEC SQL execute prep_stmt using sql descriptor 'desc1';
```

Supplying Parameters Through an sqlda Structure (ESQL/C)

You can specify the **sqlda** pointer in the USING DESCRIPTOR clause of the EXECUTE statement.

Each time the EXECUTE statement is run, the values that the descriptor structure describes are used to replace question-mark (?) placeholders in the PREPARE statement.

The **sqlda.sqld** value specifies the number of input values that are described in occurrences of **sqlvar**. This number must correspond to the number of dynamic parameters in the prepared statement.

The following example shows how to use an **sqlda** structure to execute a prepared statement in :

```
EXEC SQL execute prep_stmt using descriptor pointer2;
```

EXECUTE FUNCTION statement

Use the EXECUTE FUNCTION statement to invoke a user-defined function or a built-in routine that returns a value.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_561_Connect_42_exfu002) unique_561_Connect_42_exfu002(explicit id unique_561_Connect_42_exfu003)
unique_561_Connect_42_exfu003(explicit id unique_561_Connect_42_exfu011) unique_561_Connect_42_exfu011(explicit id unique_561_Connect_42_exfu012)
unique_561_Connect_42_exfu012(explicit id unique_561_Connect_42_exfu007) unique_561_Connect_42_exfu007(explicit id unique_561_Connect_42_exfu008)
unique_561_Connect_42_exfu008 **EXECUTE FUNCTION** { { *function* | SPL_var } ([<Argument>]) <INTO Clause> } [**WITH TRIGGER REFERENCES**]

Element	Description	Restrictions	Syntax
function	Name of a user-defined function to execute	Must be registered in the database	Database Object Name on page 1225
SPL_var	Variable that contains the name of an SPL routine to be executed	Must be a CHAR, VARCHAR, NCHAR, or NVARCHAR data type that contains the non-NULL name of an existing SPL function	Identifier on page 1231

Usage

The EXECUTE FUNCTION statement invokes a user-defined function (UDF), with arguments, and specifies where the results are to be returned.

An external C or Java™ language function returns exactly one value.

An SPL function can return one or more values.

You cannot use the EXECUTE FUNCTION statement to invoke any type of user-defined procedure that returns no value. Instead, use the EXECUTE PROCEDURE or EXECUTE ROUTINE statement to execute procedures.

You must have the Execute privilege on the user-defined function.

For more information, see GRANT statement on page 537.

In ANSI/ISO-compliant databases that support implicit transactions, the EXECUTE FUNCTION statement does not, by default, begin a new transaction. SQL statements within the invoked function, however, can begin a new transaction.

Negator Functions and Their Companions

If a UDF that returns a BOOLEAN value has a companion function, any user who executes the function must have the Execute privilege on both the function and on its companion. For example, if a function has a negator function, any user who executes the function must have the Execute privilege on both the function and its negator. In addition, the companion function must have the same owner as its negator function.

For information on how to designate a UDF as the companion to its negator function, see NEGATOR on page 1277.

How the EXECUTE FUNCTION Statement Works

For a user-defined function (UDF) to be executed with the EXECUTE FUNCTION statement, the following conditions must exist:

- The qualified function name or the function signature (the function name with its parameter list) must be unique within the name space or database.
- The function must exist in the current database.

If EXECUTE FUNCTION specifies fewer arguments than the user-defined function expects, the unspecified arguments are said to be *missing*. Missing arguments are initialized to their corresponding parameter default values, if these were defined. The syntax of specifying default values for parameters is described in Routine Parameter List on page 1281.

EXECUTE FUNCTION returns an error under the following conditions:

- EXECUTE FUNCTION specifies more arguments than the UDF expects.
- One or more arguments are missing and do not have default values.
- The fully qualified function name or the function signature is not unique.
- No function with the specified name or signature that you specify is found.
- EXECUTE FUNCTION attempts to invoke a user-defined procedure.
- In a distributed transaction, a UDR that is running on a subordinate participating server calls a remote function on a database of another server instance.

If the *function* name is not unique within the database, you must specify enough *parameter_type* information to disambiguate the name. See the section Arguments on page 1209 for more information about how to specify parameters when invoking a function.

The *specific name* of an external UDR is valid in some DDL statements, but is not valid in contexts where you invoke the function.

If Informix® cannot resolve an ambiguous function name whose signature differs from the signature of another routine only in an unnamed-ROW type parameter, an error is returned. (This error cannot be anticipated by the database server when the ambiguous function is defined.)

INTO Clause

(explicit id unique_703_Connect_42_into002) unique_703_Connect_42_into002(explicit id unique_703_Connect_42_into003) unique_703_Connect_42_into003

INTO Clause

" **INTO** { | *data_var* [[] { : | [] $ } **INDICATOR** *indicator_var*] | *data_structure* } "

Element	Description	Restrictions	Syntax
data_structure	Structure that was declared as a host variable	Individual elements of structure must be compatible with the data types of the returned values	Language specific
data_var	Variable to receive the value that a user-defined function returns	See Data Variables on page 504.	Language specific
indicator_var	Program variable to store a return code if the corresponding *data_var* receives a NULL value	Use an indicator variable if the value of the corresponding *data_var* might be NULL	Language specific

You must include an INTO clause with EXECUTE FUNCTION to specify the variables that receive the values that a user-defined function returns. If the function returns more than one value, the values are returned into the list of variables in the order in which you specify them.

If the EXECUTE FUNCTION statement stands alone (that is, it is not part of a DECLARE statement and does not use the INTO clause), it must execute a noncursor function. A noncursor function returns only one row of values. The following example shows a SELECT statement in :

```
EXEC SQL EXECUTE FUNCTION
   cust_num(fname, lname, company_name) INTO :c_num;
```

Data Variables

If you issue EXECUTE FUNCTION within an ESLQ/C program, *data_var* must be a host variable. Within an SPL routine, *data_var* must be an SPL variable.

If you issue EXECUTE FUNCTION within a CREATE TRIGGER statement, *data_var* must be a column name in the triggering table or in another table.

INTO Clause with Indicator Variables (ESQL/C)

You should use an indicator variable if the possibility exists that data returned from the user-defined function is NULL. For more information about indicator variables, see the *HCL® Informix® Enterprise Replication Guide*.

INTO Clause with Cursors

If EXECUTE FUNCTION calls a UDF that returns more than one row of values, it must execute a cursor function. A cursor function can return one or more rows of values and must be associated with a Function cursor to execute.

If the SPL function returns more than one row or a collection data type, you must access the rows or collection elements with a cursor.

To return more than one row of values, an external function (one written in the C or Java™ language) must be defined as an iterator function. For more information on iterator functions, see the *HCL® Informix® DataBlade® API Programmer's Guide*.

In an SPL routine, if a SELECT returns more than one row, you must use the FOREACH statement to access the rows individually. The INTO clause of the SELECT statement can store the fetched values. For more information, see FOREACH on page 935.

To return more than one row of values, an SPL function must include the WITH RESUME keywords in its RETURN statement. For more information on how to write SPL functions, see the *HCL® Informix® Guide to SQL: Tutorial*.

In programs, the DECLARE statement can declare a Function cursor and the FETCH statement can return rows individually from the cursor. You can put the INTO clause in the EXECUTE FUNCTION or in the FETCH statement, but you cannot put it in both. The following code examples show different ways you can use the INTO clause:

- Using the INTO clause in the EXECUTE FUNCTION statement:

```
EXEC SQL declare f_curs cursor for
   execute function get_orders(customer_num)
   into :ord_num, :ord_date;
EXEC SQL open f_curs;
while (SQLCODE == 0)
   EXEC SQL fetch f_curs;
EXEC SQL close f_curs;
```

- Using the INTO clause in the FETCH statement:

```
EXEC SQL declare f_curs cursor for
   execute function get_orders(customer_num);
EXEC SQL open f_curs;
while (SQLCODE == 0)
   EXEC SQL fetch f_curs into :ord_num, :ord_date;
EXEC SQL close f_curs;
```

Alternatives to PREPARE ... EXECUTE FUNCTION ... INTO

About this task

In ESQL/C, you cannot prepare an EXECUTE FUNCTION statement that includes the INTO clause. For similar functionality, however, follow these steps:

1. Prepare the EXECUTE FUNCTION statement with no INTO clause.
2. Declare a Function cursor for the prepared statement.
3. Open the cursor.
4. Execute the FETCH statement with an INTO clause to fetch the returned values into program variables.

Alternatively, you can do the following:

1. Declare a cursor for the EXECUTE FUNCTION statement without first preparing the statement, and include the INTO clause in the EXECUTE FUNCTION when you declare the cursor.
2. Open the cursor.
3. Fetch the returned values from the cursor without using the INTO clause of the FETCH statement.

Dynamic Routine-Name Specification of SPL Functions

Dynamic routine-name specification simplifies the writing of an SPL function that calls another SPL routine whose name is not known until runtime. To specify the name of an SPL routine in the EXECUTE FUNCTION statement, instead of listing the explicit name of an SPL routine, you can use an SPL variable to hold the routine name. For more information about how to execute SPL functions dynamically, see the *HCL® Informix® Guide to SQL: Tutorial*.

EXECUTE IMMEDIATE statement

Use the EXECUTE IMMEDIATE statement to perform tasks equivalent to what the PREPARE, EXECUTE, and FREE statements accomplish, but as a single operation.

Use this Dynamic SQL statement with and SPL.

Syntax

(explicit id unique_711_Connect_42_exim000) unique_711_Connect_42_exim000(explicit id unique_711_Connect_42_exim002)

unique_711_Connect_42_exim002 **EXECUTE IMMEDIATE** { ' *statement* [[] ; *statement*] [;] ' | *statement_var* | []*char_expr* }

Element	Description	Restrictions	Syntax
char_expr	Expression that evaluates to a character data type	Must evaluate to a CHAR, LVARCHAR, NCHAR, NVARCHAR, or VARCHAR data type	Expression on page 1003
statement	Text of a valid SQL statement	See the same sections that are listed below for *statement_var*	See this chapter.
statement _var	Variable containing *statement* or (in ESQL/C) a semicolon-separated list of statements	Must be a previously declared variable of type CHAR, NCHAR, NVARCHAR, or VARCHAR (or in SPL, LVARCHAR). See EXECUTE IMMEDIATE and Restricted Statements	Language specific

Element	Description	Restrictions	Syntax
		on page 507 and Restrictions on Valid Statements on page 508.	

Usage

The EXECUTE IMMEDIATE statement dynamically executes a single SQL statement (or in ESQL/C routines, a semicolon-separated list of SQL statements) that is constructed during program execution. For example, you can obtain the name of a database from program input, construct the DATABASE statement as a program variable, and then use EXECUTE IMMEDIATE to execute the statement, which opens the specified database.

Within ESQL/C routines, the statement text specified by the variable or quoted string can include more than one SQL statement, if consecutive statements are separated by a semicolon (;) delimiter. In SPL routines, however, only one statement can be included. The *statement* cannot be an SPL statement, but can be any SQL statement that is not listed in the sections EXECUTE IMMEDIATE and Restricted Statements on page 507 or Restrictions on Valid Statements on page 508.

The specification that follows the IMMEDIATE keyword, if valid, is parsed and executed; then all data structures and memory resources are released immediately. Unless you use EXECUTE IMMEDIATE, these operations would otherwise require separate PREPARE, EXECUTE, and FREE statements.

The session environment values (such as the EXTDIRECTIVES, OPTCOMPIND, or USELASTCOMMITTED settings of the ESQL/C or SPL routine that issues the EXECUTE IMMEDIATE statement) override the corresponding ONCONFIG parameter values, if these are different.

In ANSI/ISO-compliant databases that support implicit transactions, the EXECUTE IMMEDIATE statement does not, by default, begin a new transaction. Execution of the specified SQL statement text, however, can begin a new transaction.

EXECUTE IMMEDIATE and Restricted Statements

The EXECUTE IMMEDIATE statement cannot execute the following SQL statements.

- CLOSE
- CONNECT
- DECLARE
- DISCONNECT
- EXECUTE
- EXECUTE FUNCTION
- EXECUTE PROCEDURE
- FETCH
- FLUSH
- FREE
- GET DESCRIPTOR
- GET DIAGNOSTICS

- OPEN
- OUTPUT
- PREPARE
- PUT
- SELECT
- SET AUTOFREE
- SET CONNECTION
- SET DEFERRED_PREPARE
- SET DESCRIPTOR
- WHENEVER

For EXECUTE PROCEDURE, this restriction applies only to calls that return one or more values.

The only form of the SELECT statement that EXECUTE IMMEDIATE supports as statement text is `SELECT ... INTO TEMP table`. For the syntax of the INTO TEMP *table* clause in SELECT statements, see #unique_714.

In addition, ESQL/C cannot use the EXECUTE IMMEDIATE statement to execute the following statements in text that contains multiple SQL statements that are separated by semicolons:

- CLOSE DATABASE
- CREATE DATABASE
- DATABASE
- DROP DATABASE
- SELECT (except SELECT INTO TEMP)

The EXECUTE IMMEDIATE statement cannot process SQL statement text that includes question mark (?) symbols as placeholders. Use the PREPARE statement and either a cursor or the EXECUTE statement to execute a dynamically constructed SELECT statement.

(In SPL routines, the EXECUTE IMMEDIATE statement can execute only a single SQL statement. If the specification that immediately follows the IMMEDIATE keyword evaluates to a list of multiple SQL statements, or by a NULL value, or text that is not a valid SQL statement, the database server issues a runtime error.)

Restrictions on Valid Statements

The following restrictions apply to the statements contained in the character expression, quoted string, or statement variable that immediately follows the EXECUTE IMMEDIATE keywords:

- The SQL statement cannot contain a host-language comment.
- Names of host-language variables are not recognized as such in prepared text.

  The only identifiers that you can use are names registered in the system catalog of the current database, such as table names and column names.

- The statement cannot reference a host-variable list or a descriptor; it must not contain any question-mark (?) placeholders, which are allowed with a PREPARE statement.
- The text must not include any embedded SQL statement prefix, such as the dollar sign ($) or the keywords EXEC SQL.

  Although it is not required, the SQL statement terminator (;) can be included in the statement text.
- A SELECT or INSERT statement specified within the EXECUTE IMMEDIATE statement cannot contain a Collection-Derived Table clause.

  EXECUTE IMMEDIATE cannot process input host variables, which are required for a collection variable. Use the EXECUTE statement or a cursor to process prepared accesses to collection variables.

Handling Exceptions from EXECUTE IMMEDIATE Statements

If the parser detects a syntax error when the EXECUTE IMMEDIATE statement is compiled, it issues a compilation error, and no executable UDR is produced until the syntax is corrected and recompiled. If the parser accepts the EXECUTE IMMEDIATE syntax and the UDR compiles successfully, but an exception occurs during a call to the UDR when the EXECUTE IMMEDIATE statement is executing, the database server issues an error at runtime. Runtime errors can be trapped by the WHENEVER statement, or by some other exception-handling mechanism in the program logic of the UDR.

For routines written in the SPL language, SQL expressions are evaluated at runtime, not when the routine is compiled or optimized. If an expression that follows the IMMEDIATE keyword specifies invalid SQL statement text, Informix® issues a runtime exception, rather than a compilation error. After any runtime error condition in an SPL routine, program control passes to the ON EXCEPTION statement block (if this is defined); otherwise, execution of the UDR terminates abnormally, and an error is returned to the calling context. For information on how to handle runtime errors in SPL routines, see the descriptions of the SPL statement ON EXCEPTION on page 949. (See also the built-in SQL function **SQLCODE**.)

Examples of the EXECUTE IMMEDIATE Statement

The following ESQL/C examples show EXECUTE IMMEDIATE statements in . Both examples use host variables that contain a CREATE DATABASE statement.

```
sprintf(cdb_text1, "create database %s", usr_db_id);
EXEC SQL execute immediate :cdb_text1;

sprintf(cdb_text2, "create database %s", usr_db_id2);
EXEC SQL execute immediate :cdb_text2;
```

The next example shows an SPL program fragment that declares local SPL variables and assigns to them portions of the text of two DDL statements. It then issues an EXECUTE IMMEDIATE statement to drop a table called **DYN_TAB**, specifying the DROP TABLE statement text in an SPL variable. The second EXECUTE IMMEDIATE statement in this example creates a table of the same name, in this case specifying the CREATE TABLE statement text in a character expression that concatenates the contents of two SPL variables.

```
   CREATE PROCEDURE myproc()
   DEFINE COLS    VARCHAR(22);
   DEFINE CRTOPER VARCHAR(16);
```

```
    DEFINE DRPOPER VARCHAR(16);
    DEFINE TABNAME VARCHAR(16);
    DEFINE QRYSTR  VARCHAR(100);
    ...
    LET CRTOPER = "CREATE TABLE ";
    LET DRPOPER = "DROP TABLE ";
    LET TABNAME = "DYN_TAB";
    LET COLS = "(ID INT, NAME CHAR(20))";
    LET QRYSTR = DRPOPER || TABNAME;
    EXECUTE IMMEDIATE QRYSTR;

    EXECUTE IMMEDIATE CRTOPER || TABNAME || COLS;

    END PROCEDURE;
```

EXECUTE PROCEDURE statement

Use the EXECUTE PROCEDURE statement to invoke a user-defined procedure or a built-in routine. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_332_Connect_42_expr002) unique_332_Connect_42_expr002(explicit id unique_332_Connect_42_expr009)

unique_332_Connect_42_expr009(explicit id unique_332_Connect_42_expr006) unique_332_Connect_42_expr006 **EXECUTE PROCEDURE** { *procedure* |

[] *SPL_var* | *function* } ([<Argument>[]]) [[] **INTO** *output_var*] [[] **WITH TRIGGER REFERENCES**]

Element	Description	Restrictions	Syntax
function	SPL routine to execute	Must exist	Database Object Name on page 1225
output_var	Host variable or program variable that receives the returned value from UDR	In the context of a CREATE TRIGGER statement, must contain column names in the triggering table or in another table	Language specific
procedure	User-defined procedure to execute	Must exist	Database Object Name on page 1225
SPL_var	Variable that contains the name of the SPL routine to execute	Must be a character data type that contains the non-NULL name of an SPL routine.	Identifier on page 1231

Usage

The EXECUTE PROCEDURE statement invokes a user-defined procedure and specifies its arguments.

For compatibility with earlier Informix® versions, you can use the EXECUTE PROCEDURE statement to execute an SPL function that the CREATE PROCEDURE statement defined.

If the EXECUTE PROCEDURE statement returns more than one row, the result set must be processed within a FOREACH loop of an SPL routine, or else accessed through a cursor of an ESQL/C routine.

In ANSI/ISO-compliant databases that support implicit transactions, the EXECUTE PROCEDURE statement does not, by default, begin a new transaction. SQL statements within the invoked procedure, however, can begin a new transaction.

Causes of Errors

EXECUTE PROCEDURE returns an error under the following conditions.

- It has more arguments than the called procedure expects.
- One or more arguments that do not have default values are missing.
- The fully-qualified procedure name or the routine signature is not unique.
- No procedure with the specified name or signature is found.
- In a distributed transaction, a UDR that is running on a subordinate participating server calls a remote procedure on another server instance

For example, the following statement fails with an error:

```
EXECUTE PROCEDURE ifx_unload_module
   ("C:\usr\apps\opaque_types\circle.dll");
```

This built-in procedure requires a second argument, as explained in the topic #unique_719.

If the *procedure* name is not unique within the database, you must specify enough *parameter_type* information to disambiguate the name. See Arguments on page 1209 for additional information about how to specify parameters when invoking a procedure. (The *specific name* of an external UDR is valid in DDL statements, but is not valid in contexts where you invoke the procedure.)

Using the INTO Clause

Use the INTO clause to specify where to store the values that the SPL function returns.

If an SPL function returns more than one value, the values are returned into the list of variables in the order in which you specify them. If an SPL function returns more than one row or a collection data type, you must access the rows or collection elements with a cursor.

You cannot prepare an EXECUTE PROCEDURE statement that has an INTO clause. For more information, see Alternatives to PREPARE ... EXECUTE FUNCTION ... INTO on page 505.

The WITH TRIGGER REFERENCES Keywords

You must include the WITH TRIGGER REFERENCES keywords when you use the EXECUTE PROCEDURE statement to invoke a trigger procedure.

A trigger procedure is an SPL routine that EXECUTE PROCEDURE can invoke only from the FOR EACH ROW section of the Action clause of a trigger definition. Such procedures must include the REFERENCING clause and the FOR clause in the CREATE PROCEDURE statement that defined the procedure. This REFERENCING clause declares names for correlated variables that the procedure can use to reference the *old* column value in the row when the trigger event occurred, or the *new* value of the column after the row was modified by the trigger. The FOR clause specifies the table or view on which the trigger is defined.

Example of Invoking a Trigger Procedure

The following example defines three tables and a trigger procedure that references one of these tables in its FOR clause:

```
CREATE TABLE tab1 (col1 INT,col2 INT);
CREATE TABLE tab2 (col1 INT);
CREATE TABLE temptab1
   (old_col1 INTt, new_col1 INT, old_col2 INT, new_col2 INT);

/* The following procedure is invoked from an INSERT trigger in this example.
*/
CREATE PROCEDURE proc1()
REFERENCING OLD AS o NEW AS n FOR tab1;

IF (INSERTING) THEN  -- INSERTING Boolean operator
        LET n.col1 = n.col1 + 1;   -- You can modify new values.
        INSERT INTO temptab1 VALUES(0,n.col1,1,n.col2);
END IF

IF (UPDATING) THEN  -- UPDATING Boolean operator
       -- you can access relevant old and new values.
       INSERT INTO temptab1 values(o.col1,n.col1,o.col2,n.col2);
END IF

if (SELECTING) THEN  -- SELECTING Boolean operator
        -- you can access relevant old values.
        INSERT INTO temptab1 VALUES(o.col1,0,o.col2,0);
END IF

if (DELETING) THEN   -- DELETING Boolean operator
        DELETE FROM temptab1 WHERE temptab1.col1 = o.col1;
END IF

END PROCEDURE;
```

This example illustrates that the triggered action can be a different DML operation from the triggering event. Although this procedure inserts a row when an Insert trigger calls it, and deletes a row when a Delete trigger calls it, it also performs INSERT operations if it is called by a Select trigger or by an Update trigger.

The **proc1( )** trigger procedure in this example uses Boolean conditional operators that are valid only in trigger routines. The INSERTING operator returns true only if the procedure is called from the FOR EACH ROW action of an INSERT trigger. This procedure can also be called from other triggers whose trigger event is an UPDATE, SELECT, or DELETE. statement, because the UPDATING, SELECTING and DELETING operators return true (`t`) if the procedure is invoked in the triggered action of the corresponding type of triggering event.

The following statement defines an Insert trigger on **tab1** that calls **proc1( )** from the FOR EACH ROW section as its triggered action, and perform an INSERT operation that activates this trigger:

```
CREATE TRIGGER ins_trig_tab1 INSERT ON tab1 REFERENCING NEW AS post
  FOR EACH ROW(EXECUTE PROCEDURE proc1() WITH TRIGGER REFERENCES);
```

Note that the REFERENCING clause of the trigger declares a correlation name for the NEW value that is different from the correlation name that the trigger procedure declared. These names do not need to match, because the correlation name that

was declared in the trigger procedure has that procedure as its scope of reference. The following statement activates the **ins_trig_tab1** trigger, which executes the **proc1( )** procedure.

```
INSERT INTO tab1 VALUES (111,222);
```

Because the trigger procedure increments the new value of **col1** by 1, the value inserted is (112, 222), rather than the value that the triggering event specified.

Dynamic Routine-Name Specification of SPL Procedures

Dynamic routine-name specification simplifies the writing of an SPL routine that calls another SPL routine whose name is not known until runtime.

To specify the name of an SPL routine in the EXECUTE PROCEDURE statement, instead of listing the explicit name of an SPL routine, you can use an SPL variable to hold the routine name.

If the SPL variable names an SPL routine that returns a value (an SPL function), include the INTO clause of EXECUTE PROCEDURE to specify a *receiving variable* (or variables) to hold the value (or values) that the SPL function returns. For more information on how to execute SPL procedures dynamically, see the *HCL® Informix® Guide to SQL: Tutorial*.

FETCH statement

Use the FETCH statement to move a cursor to a new row in the active set and to retrieve the row values from memory.

Use this statement with and with SPL.

Syntax

(explicit id unique_723_Connect_42_fet002) unique_723_Connect_42_fet002(explicit id unique_723_Connect_42_fet003) unique_723_Connect_42_fet003

FETCH [{ | **NEXT** | **PRIOR** | **PREVIOUS** | **FIRST** | **LAST** | **CURRENT** | **RELATIVE** { + { *position_num_var* | *position_num* } | - *position_num* } | **ABSOLUTE** { *row_position_var* | *row_position* } }] { *cursor_id_var* | *cursor_id* } { [**USING** { **SQL DESCRIPTOR** { *' descriptor '* | *descriptor_var* } | **DESCRIPTOR***sqlda_pointer* }] | **INTO** { | *output_var* [{ **INDICATOR** | : } *indicator_var*] | *data_structure* } }

Element	Description	Restrictions	Syntax
cursor_id	Cursor to retrieve rows	Must be open	Identifier on page 1231
cursor_id_var	Host variable storing *cursor_id*	Must be character data type	Language specific
data_structure	Structure as a host variable	Must store fetched values	Language specific
descriptor	System-descriptor area	Must have been allocated	Quoted String on page 1202
descriptor_var	Host variable storing *descriptor*	Must be allocated	Language specific
indicator_var	Host variable for return code if *output_var* can be NULL value	See Using Indicator Variables on page 517.	Language specific

Element	Description	Restrictions	Syntax
output_var	Host variable for fetched value	Must store value from row	Language specific
position_num	Position relative to current row	Value `0` fetches current row	Literal Number on page 1198
position_num_var	Host variable (= *position_num)*	Value `0` fetches current row	Language specific
row_position	Ordinal position in active set	Must be an integer >1	Literal Number on page 1198
row_position_var	Host variable (= *row_ position)*	Must be `1` or greater	Language specific
sqlda_pointer	Pointer to an **sqlda** structure	Cannot begin with **$** nor **:**	See ESQL/C .

Usage

Except as noted, sections that follow describe how to use the FETCH statement in routines. For information about the more restricted syntax and semantics of the FETCH statement in SPL routines, see Fetching from Dynamic Cursors in SPL Routines on page 521.

How the database server creates, stores, and fetches members of the active set of rows depends on whether the cursor was declared as a sequential cursor or as a scroll cursor. All cursors that the FETCH statement can reference in SPL routines are sequential cursors.

In X/Open mode, if a cursor-direction value (such as `NEXT` or `RELATIVE`) is specified, a warning message is issued, indicating that the statement does not conform to X/Open standards.

FETCH with a Sequential Cursor

A sequential cursor can fetch only the next row in sequence from the active set. The only option available is the default option, NEXT. A sequential cursor can read through a table only once each time the table is opened. The following example illustrates the FETCH statement with a sequential cursor:

```
EXEC SQL FETCH seq_curs INTO :fname, :lname;
EXEC SQL FETCH NEXT seq_curs INTO :fname, :lname;
```

When the program opens a sequential cursor, the database server processes the query to the point of locating or constructing the first row of data. The goal of the database server is to tie up as few resources as possible.

Because the sequential cursor can retrieve only the next row, the database server can frequently create the active set one row at a time.

On each FETCH operation, the database server returns the contents of the current row and locates the next row. This one-row-at-a-time strategy is not possible if the database server must create the entire active set to determine which row is the first row (as would be the case if the SELECT statement included an ORDER BY clause).

FETCH with a Scroll Cursor

These examples illustrate the FETCH statement with a scroll cursor:

```
EXEC SQL fetch previous q_curs into :orders;
EXEC SQL fetch last q_curs into :orders;
EXEC SQL fetch relative -10 q_curs into :orders;
printf("Which row? ");
scanf("
EXEC SQL fetch absolute :row_num q_curs into :orders;
```

A scroll cursor can fetch any row in the active set, either by specifying an absolute row position or a relative offset. Use the following cursor-position options to specify a particular row that you want to retrieve.

Keyword

Effect

NEXT

Retrieves next row in active set

PREVIOUS

Retrieves previous row in active set

PRIOR

Retrieves previous row in active set (Synonymous with PREVIOUS.)

FIRST

Retrieves the first row in active set

LAST

Retrieves the last row in active set

CURRENT

Retrieves the current row in active set (the same row as returned by the previous FETCH statement from the scroll cursor)

RELATIVE

Retrieves *n*th row, relative to the current cursor position in the active set, where *position_num (*or *position_num_var)* supplies *n.* A negative value indicates the *n*th row prior to the current cursor position. If *position_num* = `0`, the current row is fetched.

ABSOLUTE

Retrieves *n*th row in active set, where *row_position_var* (or *row_position*) = *n .* Absolute row positions are numbered from `1`.



Tip: Do not confuse row-position values with **rowid** values. A **rowid** value is based on the position of a row in its table and remains valid until the table is rebuilt. A row-position value (a value that the ABSOLUTE keyword retrieves) is the

relative position of the row in the current active set of the cursor; the next time the cursor is opened, different rows might be selected.

How the Database Server Implements Scroll Cursors

Because it cannot anticipate which row the program will ask for next, the database server must retain all the rows in the active set until the scroll cursor closes. When a scroll cursor opens, the database server implements the active set as a temporary table, although it might not populate this table immediately.

The first time a row is fetched, the database server copies it into the temporary table as well as returning it to the program.

When a row is fetched for the second time, it can be taken from the temporary table. This scheme uses the fewest resources, in case the program abandons the query before it fetches all the rows. Rows that are never fetched are usually not copied from the database, or are saved in a temporary table.

Specifying Where Values Go in Memory

Each value from the select list of the query or the output of the executed user-defined function must be returned into a memory location. You can specify these destinations in one of the following ways:

- Use the INTO clause of a SELECT statement.
- Use the INTO clause of an EXECUTE Function (or EXECUTE PROCEDURE) statement.
- Use the INTO clause of a FETCH statement.
- Use a system-descriptor area.
- Use an **sqlda** structure.

Using the INTO Clause

If you associate a SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement with a Function cursor, the statement can contain an INTO clause to specify variables to receive the returned values. You can use this method only when you write the SELECT, EXECUTE FUNCTION, or EXECUTE PROCEDURE statement as part of the cursor declaration; see DECLARE statement on page 433. In this case, the FETCH statement cannot contain an INTO clause.

The following example uses the INTO clause of the SELECT statement to specify program variables in :

```
EXEC SQL declare ord_date cursor for
   select order_num, order_date, po_num
      into :o_num, :o_date, :o_po;
EXEC SQL open ord_date;
EXEC SQL fetch next ord_date;
```

If you prepare a SELECT statement, the SELECT *cannot* include the INTO clause so you must use the INTO clause of the FETCH statement.

When you create a SELECT statement dynamically, you cannot use an INTO clause because you cannot name host variables in a prepared statement.

If you are certain of the number and data type of values in the projection list, you can use an INTO clause in the FETCH statement. If user input generated the query, however, you might not be certain of the number and data type of values that are being selected. In this case, you must use either a system descriptor or else a pointer to an **sqlda** structure.

Using Indicator Variables

Use an indicator variable if the returned data might be null.

The *indicator_var* parameter is optional, but use an indicator variable if the possibility exists that the value of *output_var* is NULL.

If you specify the indicator variable without the INDICATOR keyword, you cannot put a blank space between *output_var* and *indicator_var*.

For information about rules for placing a prefix before the *indicator_var*, see the *HCL® Informix® Enterprise Replication Guide*.

The host variable cannot be a DATETIME or INTERVAL data type.

When the INTO Clause of FETCH is Required

When SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) omits the INTO clause, you must specify a data destination when a row is fetched.

For example, to dynamically execute a SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement, the SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) cannot include its INTO clause in the PREPARE statement. Therefore, the FETCH statement must include an INTO clause to retrieve data into a set of variables. This method lets you store different rows in different memory locations.

You can fetch into a program-array element only by using an INTO clause in the FETCH statement. If you use a program array, you must list both the array name and a specific element of the array in *data_structure.* When you are declaring a cursor, do not refer to an array element within the SQL statement.



Tip: If you are certain of the number and data type of values in the select list of the Projection clause, you can use an INTO clause in the FETCH statement.

In the following example, a series of complete rows is fetched into a program array. The INTO clause of each FETCH statement specifies an array element as well as the array name:

```
EXEC SQL BEGIN DECLARE SECTION;
   char wanted_state[2];
   short int row_count = 0;
   struct customer_t{
   {
      int    c_no;
      char   fname[15];
      char   lname[15];
   } cust_rec[100];
EXEC SQL END DECLARE SECTION;
```

```
main()
{
   EXEC SQL connect to'stores_demo';
   printf("Enter 2-letter state code: ");
   scanf ("%s", wanted_state);
   EXEC SQL declare cust cursor for
      select * from customer where state = :wanted_state;
   EXEC SQL open cust;
   EXEC SQL fetch cust into :cust_rec[row_count];
   while (SQLCODE == 0)
   {
      printf("\n%s %s", cust_rec[row_count].fname,
         cust_rec[row_count].lname);
      row_count++;
      EXEC SQL fetch cust into :cust_rec[row_count];
   }
   printf ("\n");
   EXEC SQL close cust;
   EXEC SQL free cust;
}
```

Using a System-Descriptor Area (X/Open)

You can use a system-descriptor area to store output values when you do not know the number of return values or their data types that a SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement returns at runtime. A system-descriptor area describes the data type and memory location of one or more return values, and conforms to the X/Open standards.

The keywords USING SQL DESCRIPTOR introduce the name of the system-descriptor area into which you fetch the contents of a row or the return values of a user-defined function. You can then use the GET DESCRIPTOR statement to transfer the values that the FETCH statement returns from the system-descriptor area into host variables.

The following example shows a valid FETCH…USING SQL DESCRIPTOR statement:

```
EXEC SQL allocate descriptor 'desc';
   ...
EXEC SQL declare selcurs cursor for
   select * from customer where state = 'CA';
EXEC SQL describe selcurs using sql descriptor 'desc';
EXEC SQL open selcurs;
while (1)
   {
   EXEC SQL fetch selcurs using sql descriptor 'desc';
```

You can also use an **sqlda** structure to supply parameters dynamically.

Using sqlda Structures

You can use a pointer to an **sqlda** structure to stores output values when you do not know the number of values or their data types that a SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement returns.

This structure contains data descriptors that specify the data type and memory location for one selected value. The keywords USING DESCRIPTOR introduce the name of the pointer to the **sqlda** structure.

Tip: If you know the number and data types of all values in the select list, you can use an INTO clause in the FETCH statement. For more information, see When the INTO Clause of FETCH is Required on page 517.

To specify an sqlda structure as the location of parameters:

1. Declare an **sqlda** pointer variable.
2. Use the DESCRIBE statement to fill in the **sqlda** structure.
3. Allocate memory to hold the data values.
4. Use the USING DESCRIPTOR clause of FETCH to specify the **sqlda** structure as the location into which you fetch the returned values.

The following example shows a FETCH USING DESCRIPTOR statement:

```
struct sqlda *sqlda_ptr;
...
EXEC SQL declare selcurs2 cursor for
   select * from customer where state = 'CA';
EXEC SQL describe selcurs2 into sqlda_ptr;
...
EXEC SQL open selcurs2;
while (1)
   {
   EXEC SQL fetch selcurs2 using descriptor sqlda_ptr;
   ...
```

The **sqld** value specifies the number of output values that are described in occurrences of the **sqlvar** structures of the **sqlda** structure. This number must correspond to the number of values returned from the prepared statement.

Fetching a Row for Update

The FETCH statement does not ordinarily lock a row that is fetched. Thus, another process can modify (update or delete) the fetched row immediately after your program receives it. A fetched row is locked in the following cases:

- When you set the isolation level to Repeatable Read, each row that you fetch is locked with a read lock until the cursor closes or until the current transaction ends. Other programs can also read the locked rows.
- When you set the isolation level to Cursor Stability, the current row is locked.
- In an ANSI-compliant database, an isolation level of Repeatable Read is the default; you can set it to something else.
- When you are fetching through an update cursor (one that is declared FOR UPDATE), each row you fetch is locked with a promotable lock. Other programs can read the locked row, but no other program can place a promotable or write lock; therefore, the row is unchanged if another user tries to modify it using the WHERE CURRENT OF clause of an UPDATE or DELETE statement.

When you modify a row, the lock is upgraded to a write lock and remains until the cursor is closed or the transaction ends. If you do not modify the row, the behavior of the database server depends on the isolation level you have set. The database server releases the lock on an unchanged row as soon as another row is fetched, unless you are using Repeatable Read isolation. (See SET ISOLATION statement on page 812.)

Important: You can hold locks on additional rows even when Repeatable Read isolation is not in use or is unavailable. Update the row with unchanged data to hold it locked while your program is reading other rows. You must evaluate the effect of this technique on performance in the context of your application, and you must be aware of the increased potential for deadlock.

When you use explicit transactions, be sure that a row is both fetched and modified within a single transaction; that is, both the FETCH statement and the subsequent UPDATE or DELETE statement must fall between a BEGIN WORK statement and the next COMMIT WORK statement.

Fetching from a Collection Cursor

A Collection cursor allows you to access the individual elements of collection variables. To declare a Collection cursor, use the DECLARE statement and include the Collection-Derived Table segment in the SELECT statement that you associate with the cursor. After you open the collection cursor with the OPEN statement, the cursor allows you to access the elements of the collection variable.

To fetch elements, one at a time, from a Collection cursor, use the FETCH statement and the INTO clause. The FETCH statement identifies the Collection cursor that is associated with the collection variable. The INTO clause identifies the host variable that holds the element value that is fetched from the Collection cursor. The data type of the host variable in the INTO clause must match the element type of the collection.

Suppose you have a table called **children** with the following structure:

```
CREATE TABLE children
(
   age         SMALLINT,
   name         VARCHAR(30),
   fav_colors            SET(VARCHAR(20) NOT NULL),
)
```

The following code fragment shows how to fetch elements from the **child_colors** collection variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection child_colors;
   varchar one_favorite[21];
   char child_name[31] = "marybeth";
EXEC SQL END DECLARE SECTION;
EXEC SQL allocate collection :child_colors;
/* Get structure of fav_colors column for untyped
 * child_colors collection variable */
EXEC SQL select fav_colors into :child_colors
   from children
   where name = :child_name;
/* Declare select cursor for child_colors collection
 * variable */
EXEC SQL declare colors_curs cursor for
   select * from table(:child_colors);
EXEC SQL open colors_curs;
do
{
   EXEC SQL fetch colors_curs into :one_favorite;
```

```
   ...
} while (SQLCODE == 0)
EXEC SQL close colors_curs;
EXEC SQL free colors_curs;
EXEC SQL deallocate collection :child_colors;
```

After you fetch a collection element, you can modify the element with the UPDATE or DELETE statements. For more information, see the UPDATE and DELETE statements in this document. You can also insert new elements into the collection variable with an INSERT statement. For more information, see the INSERT statement.

Checking the Result of FETCH

You can use the **SQLSTATE** variable to check the result of each FETCH statement. The database server sets the **SQLSTATE** variable after each SQL statement. If a row is returned successfully, the **SQLSTATE** variable contains the value `00000`. If no row is found, the database server sets the **SQLSTATE** code to `02000`, which indicates `no data found`, and the current row is unchanged. The following conditions set the **SQLSTATE** code to `02000`, indicating `no data found`:

- The active set contains no rows.
- You issue a FETCH NEXT statement when the cursor points to the last row in the active set or points past it.
- You issue a FETCH PRIOR or FETCH PREVIOUS statement when the cursor points to the first row in the active set.
- You issue a FETCH RELATIVE *n* statement when no *n*th row exists in the active set.
- You issue a FETCH ABSOLUTE *n* statement when no *n*th row exists in the active set.

The database server copies the **SQLSTATE** code from the **RETURNED_SQLSTATE** field of the system-diagnostics area. Client-server communication protocols of Informix®, such as SQLI and DRDA®, support **SQLSTATE** code values. For a list of these codes, and for information about how to get the message text, see Using the SQLSTATE Error Status Code on page 526. You can use the GET DIAGNOSTICS statement to examine the **RETURNED_SQLSTATE**field directly. The system-diagnostics area can also contain additional error information.

You can also use **SQLCODE** variable of the SQL Communications Area (**sqlca**) to determine the same results.

Fetching from Dynamic Cursors in SPL Routines

Use the FETCH statement in an SPL routine to retrieve the next row of the active set of a specified dynamic cursor into an ordered list of SPL variables that were declared in the same SPL routine.

Syntax

The syntax of the FETCH statement in SPL routines is a subset of the syntax that FETCH supports in routines.

FETCH *cursor_id* **INTO** *output_var*

Element	Description	Restrictions	Syntax
cursor_id	Name of a dynamic cursor	Must be open and must have been declared in the same SPL routine	Identifier on page 1231

Element	Description	Restrictions	Syntax
output_var	An SPL variable to store a fetched value from the row	Must have been declared locally or globally in the calling context, and must be of a data type compatible with the fetched column value	Identifier on page 1231

Just as in ESQL/C routines, the list of output variables must correspond in number, order, and data type with column values that the SQL statement associated with the rows returned by the specified cursor.

All SPL cursors are sequential cursors. Your UDR must include logic to detect the end of the active set of the cursor, because the NOTFOUND condition does not automatically raise an exception in SPL.

The built-in **SQLCODE** function, which can only be called from SPL routines, can return the status code of a FETCH operation.

All other restrictions of ESQL/C on FETCH statements that reference sequential Select or Function cursors also apply to FETCH operations in SPL.

The FETCH statement in SPL routines does not support the following ESQL/C features:

- cursor names specified as host variables
- positional specifications or positional keywords (which require scroll cursors)
- the USING clause with descriptors or with **sqlda** pointers.

In the SPL language, indicator variables are not needed. If the FETCH operation retrieves a NULL value, the SPL variable that receives that fetched value is set to NULL.

The FETCH statement can reference only dynamic cursors that the DECLARE statement defined. The *cursor_id* cannot specify the name of a direct cursor that the FOREACH statement of SPL declared.

FLUSH statement

Use the FLUSH statement to force rows that a PUT statement buffered to be written to the database.

Syntax

```
FLUSH { cursor_id  | cursor_id_var }
```

Element	Description	Restrictions	Syntax
cursor_id	Name of a cursor	Must have been declared	Identifier on page 1231
cursor_id_var	Host variable that holds the value of *cursor_id*	Must be a character data type	Language specific

Usage

Use this statement, which is an extension to the ANSI/ISO standard for SQL, with .

The PUT statement adds a row to a buffer, whose content is written to the database when the buffer is full. Use the FLUSH statement to force the insertion when the buffer is not full.

If the program terminates without closing the cursor, the buffer is left unflushed. Rows placed into the buffer since the last flush are lost. Do not expect the end of the program to close the cursor and flush the buffer automatically. The following example shows a FLUSH statement that operates on a cursor called **icurs**:

```
FLUSH icurs
```

Example

The following example assumes that a function named *next_cust* returns either information about a new customer or null data to signal the end of input:

```
EXEC SQL BEGIN WORK;
EXEC SQL OPEN new_custs;

while(SQLCODE == 0)
{
 next_cust();
 if(the_company == NULL)
  break;

 EXEC SQL PUT new_custs;
}

if(SQLCODE == 0)  /* if no problem with PUT */
{
 EXEC SQL FLUSH new_custs;
 /* write any rows left */

 if(SQLCODE == 0)  /* if no problem with FLUSH */
  EXEC SQL COMMIT WORK;  /* commit changes */
}
else
 EXEC SQL ROLLBACK WORK; /* else undo changes */
```

The code in this example calls *next_cust* repeatedly. When it returns non-null data, the **PUT** statement sends the returned data to the row buffer. When the buffer fills, the rows it contains are automatically sent to the database server. The loop normally ends when *next_cust* has no more data to return.

Error Checking FLUSH Statements

The SQL Communications Area (**sqlca**) structure contains information on the success of each FLUSH statement and the number of rows that are inserted successfully. The result of each FLUSH statement is described in the fields of the **sqlca**: **sqlca.sqlcode**, **SQLCODE**, and **sqlca.sqlerrd[2]**.

When you use data buffering with an Insert cursor, you do not discover errors until the buffer is flushed. For example, an input value that is incompatible with the data type of the column for which it is intended is discovered only when the buffer is flushed. When an error is discovered, any rows in the buffer that are located after the error are *not* inserted; they are lost from memory.

The **SQLCODE** field is set either to an error code or to zero (0) if no error occurs. The third element of the **SQLERRD** array is set to the number of rows that are successfully inserted into the database:

- If a block of rows is successfully inserted into the database, **SQLCODE** is set to zero (0) and **SQLERRD** to the count of rows.
- If an error occurs while the FLUSH statement is inserting a block of rows, **SQLCODE** shows which error, and **SQLERRD** contains the number of rows that were successfully inserted. (Uninserted rows are discarded from the buffer.)

Tip: When you encounter an **SQLCODE** error, a corresponding **SQLSTATE** error value also exists. Client-server communication protocols of Informix®, such as SQLI and DRDA®, support **SQLSTATE** code values. For a list of these codes, and for information about how to get the message text, see Using the SQLSTATE Error Status Code on page 526.

To count the number of rows actually inserted into the database as well as the number not yet inserted

1. Prepare two bigint variables, for example, **total** and **pending**.
2. When the cursor opens, set both variables to `0`.
3. Each time a PUT statement executes, increment both **total** and **pending**.
4. Whenever a FLUSH statement executes or the cursor is closed, subtract the third field of the **SQLERRD** array from **pending**.

FREE statement

Use the FREE statement to release resources that are allocated to a prepared statement or to a cursor.

Syntax

(explicit id unique_739_Connect_42_free001) unique_739_Connect_42_free001 **FREE** { *cursor_id* | *statement_id* | { [] | *cursor_id_var* | *statement_id_var* } }

Element	Description	Restrictions	Syntax
cursor_id	Name of a cursor	Must have been declared	Identifier on page 1231
cursor_id_var	Host variable that holds the value of cursor_id	Must be a character data type	Language specific
statement_id	Identifier of a prepared SQL statement	Must be defined in a previous PREPARE statement	PREPARE statement on page 623
statement_id_var	Host variable that stores the name of a prepared object	Must be declared as a character data type.	PREPARE statement on page 623

Usage

Use this statement, which is an extension to the ANSI/ISO standard for SQL, with or with SPL.

FREE releases the resources that the database server and (for ESQL/C) the application-development tool allocated for a prepared statement or for a cursor.

If you declared a cursor for a prepared statement, FREE *statement_id* (or *statement_id_var*) releases only the resources in the application development tool; the cursor can still be used. The resources in the database server are released only when you free the cursor.

If you prepared a statement (but did not declare a cursor for it), FREE *statement_id* (or FREE *statement_id_var*) releases the resources in both the application development tool and the database server.

After you free a statement, you cannot execute it or declare a cursor for it until you prepare it again.

The following example shows the sequence of statements that is used to free an implicitly prepared statement:

```
EXEC SQL prepare sel_stmt from 'select * from orders';
...
EXEC SQL free sel_stmt;
```

The following example shows the sequence of statements that are used to release the resources of an explicitly prepared statement. The first FREE statement in this example frees the cursor. The second FREE statement in this example frees the prepared statement.

```
sprintf(demoselect, "%s %s",
   "select * from customer ",
   "where customer_num between 100 and 200");
EXEC SQL prepare sel_stmt from :demoselect;
EXEC SQL declare sel_curs cursor for sel_stmt;
EXEC SQL open sel_curs;
...
EXEC SQL close sel_curs;
EXEC SQL free sel_curs;
EXEC SQL free sel_stmt;
```

If you declared a cursor for a prepared statement, freeing the cursor releases only the resources in the database server. To release the resources for the statement in the application-development tool, use FREE *statement_id* (or FREE *statement_id_var*). If a cursor is not declared for a prepared statement, freeing it releases the resources in both the application-development tool and the database server. For an ESQL/C example of a FREE statement that frees a cursor, see the previous example.

After a cursor is freed, it cannot be opened until it is declared again. The cursor should be explicitly closed before it is freed.

When an SPL routine completes execution, the database server automatically releases any resources that had been allocated to the cursor or to prepared statements by PREPARE or DECLARE statements in the routine, if these have not already been released by the FREE statement.

The FREE statement in SPL routines cannot reference the *cursor_id* of a direct cursor that the FOREACH statement of SPL can declare.

GET DIAGNOSTICS statement

Use the GET DIAGNOSTICS statement to return diagnostic information about the most recently executed SQL statement.

Syntax

```
(explicit id unique_749_Connect_42_gdi001) unique_749_Connect_42_gdi001(explicit id unique_749_Connect_42_gdi002) unique_749_Connect_42_gdi002 GET DIAGNOSTICS { <Statement Clause>[] | <EXCEPTION Clause>[] }
```

Usage

Use this statement with .

The GET DIAGNOSTICS statement retrieves specified status information that the database server records in a structure called the *diagnostics area*. Using GET DIAGNOSTICS does not change the contents of the diagnostics area.

The GET DIAGNOSTICS statement uses one of the following two clauses:

- The Statement clause returns count and overflow information about errors and warnings that the most recent SQL statement generates.
- The EXCEPTION clause provides specific information about errors and warnings that the most recent SQL statement generates.

Using the SQLSTATE Error Status Code

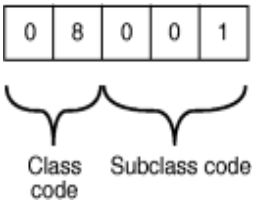
When an SQL statement executes, an error status code is automatically generated. This code represents `success`, `failure`, `warning`, or `no data found`. This error status code is stored in a built-in variable called **SQLSTATE**.

Class and Subclass Codes

The **SQLSTATE** status code is a five-character string that can contain only digits and uppercase letters.

The first two characters of the **SQLSTATE** status code indicate a class. The last three characters of the **SQLSTATE** code indicate a subclass. Figure 5: Structure of the SQLSTATE Code on page 526 shows the structure of the **SQLSTATE** code. This example uses the value `08001`, where `08` is the class code and `001` is the subclass code. The value `08001` represents the error `unable to connect with database environment`.

Figure 5. Structure of the SQLSTATE Code



The following table is a quick reference for interpreting class code values.

SQLSTATE Class Code Value

Outcome

00

Success

01

Success with warning

02

No data found

> 02

Error or warning

SQLSTATE Support for the ANSI/ISO Standard for SQL

All status codes returned to the **SQLSTATE** variable are ANSI-compliant except in the following cases:

- **SQLSTATE** codes with a class code of `01` and a subclass code that begins with an `I` are Informix®-specific warning messages.
- **SQLSTATE** codes with a class code of `IX` and any subclass code are Informix®-specific error messages.
- **SQLSTATE** codes whose class code begins with a digit in the range 5 to 9 or with an uppercase letter in the range I to Z indicate conditions that are currently undefined by the ANSI/ISO standard for SQL. The only exception is that **SQLSTATE** codes whose class code is `IX` are Informix®-specific error messages.

Client-server communication protocols of Informix®, such as SQLI and DRDA®, support these **SQLSTATE** code values.

List of SQLSTATE Codes

This table describes the class codes, subclass codes, and the meaning of all valid warning and error codes associated with the **SQLSTATE** variable.

Class	Subclass	Meaning
00	000	Success.
01	000	Success with warning.
01	002	Disconnect error. Transaction rolled back.
01	003	NULL value eliminated in set function.
01	004	String data, right truncation.
01	005	Insufficient item descriptor areas.
01	006	Privilege not revoked.
01	007	Privilege not granted.
01	I01	Database has transactions.

Class	Subclass	Meaning
01	I03	ANSI-compliant database selected.
01	I04	HCL Informix® database server selected.
01	I05	Float to decimal conversion was used.
01	I06	Informix® extension to ANSI-compliant syntax.
01	I07	UPDATE or DELETE statement does not have a WHERE clause.
01	I08	An ANSI keyword was used as a cursor name.
01	I09	Cardinalities of the projection list and of the INTO list are not equal.
01	I10	Database server running in secondary mode.
01	I11	Dataskip is turned on.
02	000	No data found.
07	000	Dynamic SQL error.
07	001	USING clause does not match dynamic parameters.
07	002	USING clause does not match target specifications.
07	003	Cursor specification cannot be executed.
07	004	USING clause is required for dynamic parameters.
07	005	Prepared statement is not a cursor specification.
07	006	Restricted data type attribute violation.
07	008	Invalid descriptor count.
07	009	Invalid descriptor index.
08	000	Connection exception.
08	001	Database server rejected the connection.
08	002	Connection name in use.
08	003	Connection does not exist.
08	004	Client unable to establish connection.
08	006	Transaction rolled back.
08	007	Transaction state unknown.
08	S01	Communication failure.
0A	000	Feature not supported.

Class	Subclass	Meaning
0A	001	Multiple server transactions.
21	000	Cardinality violation.
21	S01	Insert value list does not match column list.
21	S02	Degree of derived table does not match column list.
22	000	Data exception.
22	001	String data, right truncation.
22	002	NULL value, no indicator parameter.
22	003	Numeric value out of range.
22	005	Error in assignment.
22	027	Data exception trim error.
22	012	Division by zero (0).
22	019	Invalid escape character.
22	024	Unterminated string.
22	025	Invalid escape sequence.
23	000	Integrity constraint violation.
24	000	Invalid cursor state.
25	000	Invalid transaction state.
2B	000	Dependent privilege descriptors still exist.
2D	000	Invalid transaction termination.
26	000	Invalid SQL statement identifier.
2E	000	Invalid connection name.
28	000	Invalid user-authorization specification.
33	000	Invalid SQL descriptor name.
34	000	Invalid cursor name.
35	000	Invalid exception number.
37	000	Syntax error or access violation in PREPARE or EXECUTE IMMEDIATE.
3C	000	Duplicate cursor name.
40	000	Transaction rollback.

Class	Subclass	Meaning
40	003	Statement completion unknown.
42	000	Syntax error or access violation.
S0	000	Invalid name.
S0	001	Base table or view table already exists.
S0	002	Base table not found.
S0	011	Index already exists.
S0	021	Column already exists.
S1	001	Memory allocation failure.
IX	000	Informix® reserved error message.

Using SQLSTATE in Applications

You can use a built-in variable, called **SQLSTATE**, which you do not need to declare in your program. **SQLSTATE** contains the status code, essential for error handling, which is generated every time your program executes an SQL statement. **SQLSTATE** is created automatically. You can examine the **SQLSTATE** variable to determine whether an SQL statement was successful. If the **SQLSTATE** variable indicates that the statement failed, you can execute a GET DIAGNOSTICS statement to obtain additional error information.

For an example of how to use an **SQLSTATE** variable in a program, see Using GET DIAGNOSTICS for Error Checking on page 536.

Statement Clause

Statement Clause

```
" status_var = { ROW_COUNT | NUMBER | MORE } "
```

Element	Description	Restrictions	Syntax
status_var	Host variable to receive status information about the most recent SQL statement for the specified status field name	Must match data type of the field	Language specific

When retrieving count and overflow information, GET DIAGNOSTICS can deposit the values of the three statement fields into a corresponding host variable. The host-variable data type must be the same as that of the requested field. The following keywords represent these three fields.

Field Name Keyword	Field Data Type	Field Contents	ESQL/C Host Variable Data Type
MORE	Character	Y or N	char[2]
NUMBER	Integer	1 to 35,000	int
ROW_COUNT	Integer	0 to 999,999,999	int

Using the MORE Keyword

Use the MORE keyword to determine if the most recently executed SQL statement resulted in the following actions by the database server:

- Stored all the exceptions that it detected in the diagnostics area

  If so, GET DIAGNOSTICS returns a value of N.

- Detected more exceptions than it stored in the diagnostics area

  If so, GET DIAGNOSTICS returns a value of `Y`. (The value of MORE is always `N`.)

Using the ROW_COUNT Keyword

The ROW_COUNT keyword returns the number of rows the most recently executed DML statement processed. ROW_COUNT counts these rows:

- Inserted into a table
- Updated in a table
- Deleted from a table

Using the NUMBER Keyword

The NUMBER keyword returns the number of exceptions that the most recently executed SQL statement raised. The NUMBER field can hold a value from `1` to `35,000`, depending on how many exceptions are counted.

EXCEPTION Clause

Exception Clause

"EXCEPTION"

"{ *exception_num* | *exception_var* }"

"*information* = { CLASS_ORIGIN | CONNECTION_NAME | INFORMIX_SQLCODE | MESSAGE_LENGTH | MESSAGE_TEXT | RETURNED_SQLSTATE | SERVER_NAME | SUBCLASS_ORIGIN }"

Element	Description	Restrictions	Syntax
exception_num	Number of exceptions	Integer in range `1` to `35,000`	Literal Number on page 1198
exception_var	Variable storing *exception_num*	Must be SMALLINT or INT	Language specific
information	Host variable to receive the value of a specified exception field	Data type must match that of the specified field	Language specific

The *exception_num* literal indicates one of the exception values from the number of exceptions that the NUMBER field in the Statement clause returns.

When retrieving exception information, GET DIAGNOSTICS writes the values of each of the seven fields into corresponding host variables. These fields are located in the diagnostics area and are derived from an exception raised by the most recent SQL statement.

The host-variable data type must be the same as that of the requested field. The following table describes the seven exception information fields.

Field Name Keyword	Field Data Type	Field Contents	ESQL/C Host Variable Data Type
RETURNED_SQLSTATE	Character	**SQLSTATE** value	char[6]
INFORMIX_SQLCODE	Integer	Informix®-specific status code	int4
CLASS_ORIGIN	Character	String	char[255]
SUBCLASS_ORIGIN	Character	String	char[255]
MESSAGE_TEXT	Character	String	char[255]
MESSAGE_LENGTH	Integer	Numeric value	int
SERVER_NAME	Character	String	char[255]
CONNECTION_NAME	Character	String	char[255]

The application specifies the exception by number, using either an unsigned integer or an integer host variable (an exact numeric with a scale of 0). An exception with a value of `1` corresponds to the **SQLSTATE** value set by the most recent SQL statement other than GET DIAGNOSTICS. The association between other exception numbers and other exceptions raised by that SQL statement is undefined. Thus, no set order exists in which the diagnostic area can be filled with exception values. You always get at least one exception, even if the **SQLSTATE** value indicates success.

If an error occurs within the GET DIAGNOSTICS statement (that is, if an invalid exception number is requested), the Informix® internal **SQLCODE** and **SQLSTATE** variables are set to the value of that exception. In addition, the GET DIAGNOSTICS fields are undefined.

Using the RETURNED_SQLSTATE Keyword

The RETURNED_SQLSTATE keyword returns the **SQLSTATE** value that describes the exception.

Using the INFORMIX_SQLCODE Keyword

The INFORMIX_SQLCODE keyword returns the Informix®-specific status code. The same value is also available in the global **SQLCODE** variable. For more information, see the discussion of the **SQLCODE** variable in the *HCL® Informix® Enterprise Replication Guide*.

Using the CLASS_ORIGIN Keyword

Use the CLASS_ORIGIN keyword to retrieve the class portion of the RETURNED_SQLSTATE value. If the ISO standard for SQL defines the class, the value of CLASS_ORIGIN is equal to ISO 9075. Otherwise, the value returned by CLASS_ORIGIN is defined by Informix® and cannot be ISO 9075. The terms ANSI SQL and ISO SQL are synonymous.

Using the SUBCLASS_ORIGIN Keyword

The SUBCLASS_ORIGIN keyword returns data on the RETURNED_SQLSTATE subclass. (This value is `ISO 9075` if the ISO standard defines the subclass.)

Using the MESSAGE_TEXT Keyword

The MESSAGE_TEXT keyword returns the message text of the exception (for example, an error message).

Using the MESSAGE_LENGTH Keyword

The MESSAGE_LENGTH keyword returns the length in bytes of the current message text string.

Using the SERVER_NAME Keyword

The SERVER_NAME keyword returns the name of the database server associated with a CONNECT or DATABASE statement. GET DIAGNOSTICS updates the SERVER_NAME field when any of the following events occur:

- A CONNECT statement successfully executes.
- A SET CONNECTION statement successfully executes.
- A DISCONNECT statement successfully terminates the current connection.
- A DISCONNECT ALL statement fails.

The SERVER_NAME field is not updated, however, after these events:

- A CONNECT statement fails.
- A DISCONNECT statement fails (but this does not include the DISCONNECT ALL statement).
- A SET CONNECTION statement fails.

The SERVER_NAME field retains the value set in the previous SQL statement. If any of the preceding conditions occur on the first SQL statement that executes, the SERVER_NAME field is blank.

The Contents of the SERVER_NAME Field

The SERVER_NAME field contains different information after you execute the following statements.

Executed Statement

SERVER_NAME Field Contents

CONNECT

Contains the name of the database server to which you connect or fail to connect. Field is blank if you do not have a current connection or if you make a default connection.

DATABASE

Contains the name of the database server on which the specified database resides.

DISCONNECT

Contains the name of the database server from which you disconnect or fail to disconnect. If you disconnect and then you execute a DISCONNECT statement for a connection that is not current, the SERVER_NAME field remains unchanged.

DISCONNECT ALL

Sets the field to blank if the statement executes successfully. If the statement fails, SERVER_NAME contains the names of all the database servers from which you did not disconnect. (This information does not mean that the connection still exists.)

SET CONNECTION

Contains the name of the database server to which you switch or fail to switch

If CONNECT succeeds, SERVER_NAME is set to one of the following values:

- The **INFORMIXSERVER** value (if the connection is to a default database server, because CONNECT specified no database server)
- The name of the database server (if the connection is to a specific database server)

Using the CONNECTION_NAME Keyword

Use the CONNECTION_NAME keyword to return the name of the connection specified in your CONNECT or SET CONNECTION statement.

When the CONNECTION_NAME Keyword Is Updated

GET DIAGNOSTICS updates the CONNECTION_NAME field when the following situations occur:

- A CONNECT statement successfully executes.
- A SET CONNECTION statement successfully executes.

- A DISCONNECT statement successfully executes at the current connection.

  GET DIAGNOSTICS fills the CONNECTION_NAME field with blanks because no current connection exists.
- A DISCONNECT ALL statement fails.

When the CONNECTION_NAME Is Not Updated

The CONNECTION_NAME field is not updated in the following cases:

- A CONNECT statement fails.
- A DISCONNECT statement fails (but this does not include the DISCONNECT ALL statement).
- A SET CONNECTION statement fails.

The CONNECTION_NAME field retains the value set in the previous SQL statement. If any of the preceding conditions occurs on the first SQL statement that executes, the CONNECTION_NAME field is blank.

An implicit connection has no name. After a DATABASE statement successfully creates an implicit connection, the CONNECTION_NAME field is blank.

The contents of the CONNECTION_NAME field

The CONNECTION_NAME field contains connection information that depends on the previously executed CONNECT, SET CONNECTION, DISCONNECT, or DISCONNECT ALL statement.

The CONNECTION_NAME field contains different information after you execute the following statements.

Executed Statement

CONNECTION_NAME Field Contents

CONNECT

Contains the name, as specified in the CONNECT statement, of the connection to which you connect or fail to connect. The field is blank for no current connection or a default connection.

SET CONNECTION

Contains the name, as specified in the SET CONNECTION statement, of the connection to which you switch or fail to switch.

DISCONNECT

Contains the name, as specified in the DISCONNECT statement, of the connection from which you disconnect or fail to disconnect. If you disconnect successfully, and then execute a DISCONNECT statement for a connection that is not current, the CONNECTION_NAME field remains unchanged.

DISCONNECT ALL

Contains no information if the DISCONNECT ALL statement executes successfully. If the statement does not execute successfully, the CONNECTION_NAME field contains the names of all the connections specified in your

CONNECT statements from which you did not disconnect. This information does not mean, however, that the connection still exists.

If CONNECT is successful, CONNECTION_NAME takes one of these values:

- The name of the database environment as specified in the CONNECT statement if the CONNECT statement does not include the AS clause
- The name of the connection (the identifier that was declared after the AS keyword) if the CONNECT statement includes the AS clause

Using GET DIAGNOSTICS for Error Checking

GET DIAGNOSTICS returns values from various fields in the diagnostics area. For each field in the diagnostics area that you wish to access, you must supply a host variable of a compatible data type.

The following example illustrates how to use the GET DIAGNOSTICS statement to display error information. The example shows the error display routine called **disp_sqlstate_err( )**:

```
void disp_sqlstate_err()
{
int j;
EXEC SQL BEGIN DECLARE SECTION;
    int exception_count;
    char overflow[2];
    int exception_num=1;
    char class_id[255];
    char subclass_id[255];
    char message[255];
    int messlen;
    char sqlstate_code[6];
    int i;
EXEC SQL END DECLARE SECTION;
    printf("---------------------------------");
    printf("-------------------------\n");
    printf("SQLSTATE:
    printf("SQLCODE: %d\n", SQLCODE);
    printf("\n");
    EXEC SQL get diagnostics :exception_count = NUMBER,
        :overflow = MORE;
    printf("EXCEPTIONS:  Number=%d\t", exception_count);
    printf("More? %s\n", overflow);
    for (i = 1; i <= exception_count; i++)
    {
        EXEC SQL get diagnostics  exception :i
            :sqlstate_code = RETURNED_SQLSTATE,
            :class_id = CLASS_ORIGIN, :subclass_id = SUBCLASS_ORIGIN,
            :message = MESSAGE_TEXT, :messlen = MESSAGE_LENGTH;
        printf("- - - - - - - - - - - - - - - - - - - - -\n");
        printf("EXCEPTION %d: SQLSTATE=%s\n", i, sqlstate_code);
        message[messlen-1] ='\0';
        printf("MESSAGE TEXT: %s\n", message);
        j = stleng(class_id);
        while((class_id[j] == '\0') ||
              (class_id[j] == ' '))
            j--;
```

```
        class_id[j+1] = '\0';
        printf("CLASS ORIGIN:
        j = stleng(subclass_id);
        while((subclass_id[j] == '\0') ||
              (subclass_id[j] == ' '))
            j--;
        subclass_id[j+1] = '\0';
        printf("SUBCLASS ORIGIN:
    }
    printf("---------------------------------");
    printf("-------------------------\n");
}
```

GRANT statement

Use the GRANT statement to assign access privileges and roles to users and to other roles. Users who hold the DBSECADM role can use this statement to assign user security labels and exemptions from label-based access control (LBAC) security rules.

Syntax

(explicit id unique_312_Connect_42_revo001a) unique_312_Connect_42_revo001a(explicit id unique_312_Connect_42_revo004) unique_312_Connect_42_revo004(explicit id unique_312_Connect_42_revo006) unique_312_Connect_42_revo006(explicit id unique_312_Connect_42_revo007) unique_312_Connect_42_revo007(explicit id unique_312_Connect_42_revo008) unique_312_Connect_42_revo008(explicit id unique_312_Connect_42_revo013) unique_312_Connect_42_revo013(explicit id unique_312_Connect_42_revo009) unique_312_Connect_42_revo009(explicit id unique_312_Connect_42_revo010) unique_312_Connect_42_revo010(explicit id unique_312_Connect_42_revo011) unique_312_Connect_42_revo011(explicit id unique_312_Connect_42_revo012) unique_312_Connect_42_revo012

```
GRANT { { [ | { <Database-Level Privileges> ] | DEFAULT ROLE <Role Name> [ } TO { PUBLIC | { | 'user' } }
| { <Role Name> [ <TO Options> | <Security Administration Options> [ | <Surrogate User Properties> [ } } | { <Table-Level
Privileges> [ | { [ | <Routine-Level Privileges> [ | { { <Language-Level Privileges> [ | <Type-Level Privileges> [ | <Sequence-
Level Privileges> [ } } } } } <TO Options> }
```

TO Options

```
" TO { { | 'user' } [ WITH GRANT OPTION ] | { | 'role' | 'user' | PUBLIC } } [ AS 'grantor' ] "
```

Element	Description	Restrictions	Syntax
grantor	Authorization identifier of a user who can use REVOKE to cancel the effects of this GRANT statement. If AS clause is omitted, default is login name of user issuing this statement	Must be valid *user* name (not a *role* name). On Windows™, the *user* name cannot exceed 20 bytes. On other platforms,	Owner name on page 1262

Element	Description	Restrictions	Syntax
		the limit is 32 bytes.	
role	Name of an existing role to which you grant one or more access privileges, or to which you assign another role	Must exist in the database	Owner name on page 1262
user	Authorization identifier of a user to whom you grant one or more access privileges, or to whom you assign a role	Same as for *grantor*	Owner name on page 1262

Usage

The GRANT statement extends to other users specific discretionary access privileges or LBAC labels and exceptions that would normally accrue only to the DBA or to the creator of an object. Subsequent GRANT statements do not affect privileges that have already been granted to a user.

You can use the GRANT statement for operations like the following:

- Authorize others to use or administer a database that you create
- Allow others to view, alter, or drop a table, synonym, view or a sequence object that you create
- Allow others to use a data type or the SPL language, or to execute a user-defined routine (UDR) that you create
- Assign a role and its privileges to users, or to PUBLIC, or to another role
- Assign a default role to one or more users or to PUBLIC
- If you hold the DBSECADM role, assign LBAC security labels or exemptions from rules of LBAC security policies to users,

You can grant privileges to a previously created role or to a built-in role. You can grant a role to PUBLIC, to individual users, or to another role.

If you enclose *grantor*, *role*, or *user* in quotation marks, the name is case sensitive and is stored exactly as you typed it. In an ANSI-compliant database, if you do not use quotation marks as delimiters, the name is stored in uppercase letters.

On Windows™ only, the database server does not support *user* name that consists of more than 20 characters.

Privileges that you grant remain in effect until you cancel them with a REVOKE statement. Only the grantor of a privilege can revoke that privilege. The grantor is the person who issues the GRANT statement, unless the AS *grantor* clause transfers the right to revoke those privileges to another user.

Only the owner of an object or a user to whom privileges were explicitly granted with the WITH GRANT OPTION keywords can grant privileges on an object. Having DBA privileges is not sufficient. As DBA, however, you can grant a privilege on behalf of another user by using the AS *grantor* clause. For privileges on database objects whose owner is not a user recognized by the operating system (for example, user **informix**), the AS *grantor* clause is useful.

The keyword PUBLIC extends the specified privilege or role to the PUBLIC group of all users who connect to the database. If you intend to restrict privileges that PUBLIC already holds to only a subset of users, you must first revoke those privileges from PUBLIC.

To grant privileges on one or more fragments of a table that has been fragmented by expression, see GRANT FRAGMENT statement on page 573.

Database-Level Privileges

Database-level access privileges affect access to a database. Only individual users, rather than roles, can hold database privileges.

Database-Level Privileges

```
"{ CONNECT | RESOURCE | DBA }"
```

When you create a database with the CREATE DATABASE statement, you are the owner and automatically receive all database-level privileges.

The database remains inaccessible to any other users until you, as DBA, grant database privileges to them.

As database owner, you also receive table-level privileges on all tables in the database automatically. For more information about table-level privileges, see Table-Level Privileges on page 540.



Recommendation: Only user **informix** can modify system catalog tables directly. Except as noted specifically in your database server documentation, however, do not use DML statements to insert, delete, or update rows of system catalog tables directly, because modifying data in these tables can destroy the integrity of the database.

When database-level privileges conflict with table-level privileges, the more restrictive privileges take precedence.

Database access levels are, from lowest to highest, Connect, Resource, and DBA. Use the corresponding keyword to grant a level of access privilege.

Privilege	Effect
CONNECT	Lets you query and modify data You can modify the database schema if you own the database object that you intend to modify. Any user with the Connect privilege can perform the following operations: • Connect to the database with the CONNECT statement or another connection statement • Execute SELECT, INSERT, UPDATE, and DELETE statements, provided the user has the necessary table-level privileges • Create views, provided the user has the Select privilege on the underlying tables • Create synonyms • Create temporary tables and create indexes on the temporary tables

Privilege	Effect
	• Alter or drop a table or an index, provided the user owns the table or index (or has Alter, Index, or References privileges on the table) • Grant privileges on a table or view, provided the user owns the table (or was given privileges on the table with the WITH GRANT OPTION keywords)
RESOURCE	Lets you extend the structure of the database In addition to the capabilities of the Connect privilege, the holder of the Resource privilege can perform the following functions: • Create new tables • Create new indexes • Create new UDRs • Create new data types
DBA	Has all the capabilities of the Resource privilege and can perform the following additional operations: • Grant any database-level privilege, including the DBA privilege, to another user • Grant any table-level privilege to another user or to a role • Grant a role to a user or to another role • Revoke a privilege whose grantor you specify as the *revoker* in the AS clause of the REVOKE statement • Restrict the Execute privilege to DBAs when registering a UDR • Execute the SET SESSION AUTHORIZATION statement • Create any database object • Create tables, views, and indexes, designating another user as owner of these objects • Alter, drop, or rename database objects, regardless of who owns them • Execute the DROP DISTRIBUTIONS option of the UPDATE STATISTICS statement • Execute DROP DATABASE and RENAME DATABASE statements

User **informix** has the privilege required to alter the tables of the system catalog, including the **systables** table.

The following example uses the PUBLIC keyword to grant the Connect privilege on the currently active database to all users:

```
GRANT CONNECT TO PUBLIC;
```

You cannot grant database-level privileges to a role. Only individual users or PUBLIC can hold database-level privileges.

Table-Level Privileges

When you create a table with the CREATE TABLE statement, you are the table owner and automatically receive all table-level privileges. You cannot transfer ownership to another user, but you can grant table-level privileges to another user or to a role. (See, however, RENAME TABLE statement on page 648, which can change both the name and the ownership of a table.)

A user with the database-level DBA privilege automatically receives all table-level privileges on every table in that database.

(explicit id unique_78_Connect_42_tlp003) unique_78_Connect_42_tlp003

Table-Level Privileges

" { **ALL PRIVILEGES** | { | **INSERT** | **DELETE** | { **UPDATE** | { [] | **SELECT** | **REFERENCES** } } [(*column*)] [] | { [] | **ALTER** | **INDEX** | **UNDER** } } } "

" **ON** "

" [*owner* .] "

" { *table* | *view* | *synonym* } "

Element	Description	Restrictions	Syntax
column	Column on which the References, Select, or Update privilege is granted. Default scope is all columns of *table*, *view*, or *synonym*.	Must be a column of the *table*, *view*, or *synonym*	Identifier on page 1231
owner	Name of the user who owns the *table*, *view*, or *synonym*	Must be a valid authorization identifier	Owner name on page 1262
synonym, *table*, *view*	Synonym, table, or view on which privileges are granted.	Must exist in the current database	Identifier on page 1231

The GRANT statement can list one or more of the following keywords to specify the table privileges that you grant to the same users or roles.

Privilege	Effect
INSERT	Lets you insert rows
DELETE	Lets you delete rows
SELECT	Lets you access any column in SELECT statements. You can restrict the Select privilege to one or more columns by listing the columns.
UPDATE	Lets you access any column in UPDATE statements. You can restrict the Update privilege to one or more columns by listing the columns.
REFERENCES	Lets you define referential constraints on columns. You must have the Resource privilege to take advantage of the References privilege. (You can add, however, a referential constraint during an ALTER TABLE statement without holding the Resource privilege on the database.) You need only the References privilege to indicate cascading deletes. You do not need the Delete privilege to place

Privilege	Effect
	cascading deletes on a table. You can restrict the References privilege to one or more columns by listing the columns.
INDEX	Lets you create permanent indexes. You must have the Resource privilege to use the Index privilege. (Any user with the Connect privilege can create an index on temporary tables.)
ALTER	Lets you add or delete columns, modify column data types, add or delete constraints, change the locking mode of the table from PAGE to ROW, or add or drop a corresponding ROW data type for your table. It also lets you enable or disable indexes, constraints and triggers, as described in SET Database Object Mode statement on page 784. You must have the Resource privilege to use the Alter privilege. In addition, you also need the Usage privilege for any user-defined data type affected by the ALTER TABLE statement.
UNDER	Lets you create sub-tables under a typed table.
ALL	Provides all privileges listed above. The PRIVILEGES keyword is optional.

You can narrow the scope of a Select, Update, or References privilege by specifying the columns to which the privilege applies.

Specify the keyword PUBLIC as *user* if you intend the GRANT statement to apply to all users.

Some simple examples that follow illustrate how to give table-level privileges with the GRANT statement.

The following statement grants the privilege to delete and select values in any column in the table **customer** to users **mary** and **john**. It also grants the Update privilege, but only for columns **customer_num**, **fname**, and **lname**:

```
GRANT DELETE, SELECT, UPDATE (customer_num, fname, lname)
   ON customer TO mary, john;
```

To grant the same privileges as those above to all authorized users, use the keyword PUBLIC as the following example shows:

```
GRANT DELETE, SELECT, UPDATE (customer_num, fname, lname)
   ON customer TO PUBLIC;
```

For example, suppose a user named **mary** has created a typed table named **tab1**. By default, only user **mary** can create subtables under the **tab1** table. If **mary** wants to grant the ability to create subtables under the **tab1** table to a user named **john**, **mary** must enter the following GRANT statement:

```
GRANT UNDER ON tab1 TO john;
```

After receiving the Under privilege on table **tab1**, user **john** can create one or more subtables under **tab1**.

Effect of the ALL Keyword

The ALL keyword grants all possible table-level privileges to the specified user. If any or all of the table-level privileges do not exist for the grantor, the GRANT statement with the ALL keyword succeeds (in the sense of **SQLCODE** being set to zero,

even if the possible privileges are an empty set for the grantor on the table). In this case, however, the following **SQLSTATE** warning is returned:

```
01007 - Privilege not granted.
```

For example, assume that user **ted** has the Select and Insert privileges on the **customer** table with the authority to grant those privileges to other users.

User **ted** wants to grant all table-level privileges to user **tania**. So user **ted** issues the following GRANT statement:

```
GRANT ALL ON customer TO tania;
```

This statement executes successfully but returns **SQLSTATE** code `01007` for the following reasons:

- The statement succeeds in granting the Select and Insert privileges to user **tania** because user **ted** has those privileges and the right to grant those privileges to other users.
- The other privileges implied by the ALL keyword were not granted by user **ted** and, therefore, were not granted to user **tania**.

If you grant all table-level privileges with the ALL keyword, the privileges includes the Under privilege only if the table is a typed table. The grant of ALL privileges does not include the Under privilege if the table is not based on a ROW type.

If the table owner grants ALL privileges on a traditional relational table and later changes that table to a typed table, the table owner must explicitly grant the Under privilege to allow other users to create subtables under it.

Table Reference

You grant table-level privileges directly by specifying the name or an existing synonym of a table or of a view, which you can qualify with the *owner* name.

Table Reference

" [*owner* .] "

" { *view* | *table* | *synonym* } "

Element	Description	Restrictions	Syntax
owner	Name of the user who owns the *table*, *view*, or *synonym*	Must be a valid authorization identifier	Owner name on page 1262
synonym, *table*, *view*	Synonym, table, or view on which privileges are granted	The *table*, *view*, or *synonym* must exist in the database	Identifier on page 1231

The object on which you grant privileges must reside in the current database.

For table objects that the CREATE EXTERNAL TABLE statement has registered in the current database, the Select privilege and the Insert privilege are supported, but no other table or column access privileges can be granted or revoked.

In an ANSI-compliant database, if *owner* is not enclosed between quotation marks, the database stores the owner name in lowercase letters.

Privileges on Tables and Synonyms

In an ANSI-compliant database, if you create a table, only you, its owner, have any table-level privileges until you explicitly grant them to others.

When you create a table in a database that is *not* ANSI compliant, however, PUBLIC receives Select, Insert, Delete, Under, and Update privileges for that table and its synonyms. (The **NODEFDAC** environment variable, when set to `yes`, prevents PUBLIC from automatically receiving these table-level privileges.)

To allow access only to some users, or only on some columns in a database that is not ANSI compliant, you must explicitly revoke the privileges that PUBLIC receives by default, and then grant only the privileges that you intend. For example, this series of statements grants privileges on the entire **customer** table to users **john** and **mary**, but restricts PUBLIC access to the Select privilege on only four of the columns in that table:

```
REVOKE ALL ON customer FROM PUBLIC;
GRANT ALL ON customer TO john, mary;
GRANT SELECT (fname, lname, company, city) ON customer TO PUBLIC;
```

Privileges on a View

You must have at least the Select privilege on a table or columns to create a view on that table. For views that reference only tables in the current database, if the owner of a view loses the Select privilege on any base table underlying the view, the view is dropped.

You have the same privileges for the view that you have for the table or tables contributing data to the view. For example, if you create a view from a table to which you have only Select privileges, you can select data from your view but you cannot delete or update data. For information on how to create a view, see CREATE VIEW statement on page 421.

When you create a view, PUBLIC does not automatically receive any privileges for a view that you create. Only you have access to table data through that view. Even users who have privileges on the base table of the view do not automatically receive privileges for the view.

You can grant (or revoke) privileges on a view only if you are the owner of the underlying base tables, or if you received these privileges on the base table with the right to grant them (specified by the WITH GRANT OPTION keywords). You must explicitly grant those privileges within your authority, because PUBLIC does not automatically receive any privileges on a view when it is created.

The creator of a view can explicitly grant Select, Insert, Delete, and Update privileges for the view to other users or to a role. You cannot grant Index, Alter, Under, or References privileges on a view (nor can you specify the ALL keyword for views, because ALL confers Index, References, and Alter privileges).

When a GRANT or REVOKE statement changes the discretionary access privileges on any table referenced in the definition of an existing view, the database server does not automatically apply those privilege modifications to the view. To apply the new table access privileges to a view that depends on that table, you can use the DROP VIEW and CREATE VIEW statements to drop and recreate the view.

In this case, if the definitions of other views reference the view that you drop, or if INSTEAD OF triggers are defined on that view, you can also use CREATE VIEW and CREATE TRIGGER statements to recreate, respectively, the dependent views and the INSTEAD OF triggers that the DROP VIEW statement destroyed.

Type-Level Privileges

You can specify two privileges on data types that are not built-in data types:

- The Usage privilege on a user-defined data type
- The Under privilege on a named ROW type

Type-Level Privileges

" { **USAGE ON TYPE***type_name* | **UNDER ON TYPE***row_type_name* } "

Element	Description	Restrictions	Syntax
row_type_name	Named ROW type on which the Under privilege is granted	Named ROW data type must exist	Identifier on page 1231; Data Type on page 985
type_name	User-defined type on which the Usage privilege is granted	User-defined data type must exist.	Identifier on page 1231; Data Type on page 985

To see what privileges exist on user-defined data types, check the **sysxtdtypes** system catalog table for the *owner* of each UDT, and the **sysxtdtypeauth** system catalog table for any other users or roles that hold privileges on UDTs. See the *HCL® Informix® Guide to SQL: Reference* for information on the system catalog tables.

For all the *built-in data types*, however, these access privileges are automatically available to PUBLIC and cannot be revoked.

USAGE Privilege

You own any user-defined data type (UDT) that you create. As owner, you automatically receive the Usage privilege on that data type and can grant the Usage privilege to others so that they can reference the type name or data of that type in SQL statements. DBAs can also grant the Usage privilege for UDTs.

The following example grants user **mark** access privileges to use the **widget** user-defined type:

```
GRANT USAGE ON TYPE widget TO mark;
```

If you grant Usage privilege to a user (or to a role) that has Alter privileges, that grantee can add a column to a table that contains values of your UDT.

Without privileges from the GRANT statement, any user can issue SQL statements that reference built-in data types. In contrast, a user must receive an explicit Usage privilege from a GRANT statement to use a distinct data type, even if the distinct type is based on a built-in type.

For more information about user-defined types, see CREATE OPAQUE TYPE statement on page 263, CREATE DISTINCT TYPE statement on page 199, the discussion of data types in the *HCL® Informix® Guide to SQL: Reference* and the *IBM® Informix® Database Design and Implementation Guide*.

UNDER Privilege

You own any named ROW type that you create. If you want other users to be able to create subtypes under this named ROW type, you must grant to these users the Under privilege on your named ROW type.

For example, suppose that you create a ROW type named **rtype1**:

```
CREATE ROW TYPE rtype1 (cola INT, colb INT);
```

If you want another user named **kathy** to be able to create a subtype under this named ROW type, you must grant the Under privilege on this named ROW type to user **kathy**:

```
GRANT UNDER ON ROW TYPE rtype1 TO kathy;
```

Now user **kathy** can create another ROW type under the **rtype1** ROW type, even though **kathy** is not the owner of the **rtype1** ROW type:

```
CREATE ROW TYPE rtype2 (colc INT, cold INT) UNDER rtype1;
```

For more about named ROW types, see CREATE ROW TYPE statement on page 286, and the discussion of data types in the *HCL® Informix® Guide to SQL: Reference* and the *IBM® Informix® Database Design and Implementation Guide*.

Routine-Level Privileges

When you create a user-defined routine (UDR), you become owner of the UDR and you automatically receive the Execute privilege on that UDR.

The Execute privilege allows you to invoke the UDR with an EXECUTE FUNCTION or EXECUTE PROCEDURE statement, whichever is appropriate, or with a CALL statement in an SPL routine. The Execute privilege also allows you to use a user-defined function in an expression, as in this example:

```
SELECT * FROM table WHERE in_stock(partnum) < 20;
```

For users, roles, or members of the PUBLIC group who need the Execute privilege on a given UDR, the GRANT statement supports the following syntax:

(explicit id unique_773_Connect_42_rlp002) unique_773_Connect_42_rlp002(explicit id unique_773_Connect_42_rlp003) unique_773_Connect_42_rlp003

Routine-Level Privileges

" **EXECUTE ON** "

" { *SPL_routine* | { **PROCEDURE** | **FUNCTION** | **ROUTINE** } *routine* ([<Routine Parameter List>]) | **SPECIFIC** { **ROUTINE** | **FUNCTION** | **PROCEDURE** } <Specific Name> } "

Element	Description	Restrictions	Syntax
routine	A user-defined routine	Must exist	Identifier on page 1231
SPL_routine	An SPL routine	Must be unique in the database	Identifier on page 1231

The following statement grants Execute privilege on the **delete_order** routine to user **finn**:

```
GRANT EXECUTE ON ROUTINE delete_order TO finn;
```

Whether you must grant the Execute privilege explicitly depends on the following conditions:

- If you have DBA-level privileges, you can use the DBA keyword of CREATE FUNCTION or CREATE PROCEDURE to restrict the default Execute privilege to users with the DBA privilege. You must explicitly grant the Execute privilege on that UDR to users who do not have the DBA privilege.
- If you have the Resource database-level privilege but not the DBA privilege, you cannot use the DBA keyword when you create a UDR:
  - When you create a UDR in a database that is *not* ANSI compliant, PUBLIC can execute that UDR. You do not need to issue a GRANT statement for other users to receive the Execute privilege.
  - Setting the **NODEFDAC** environment variable to `yes` prevents PUBLIC from executing the UDR until you explicitly grant the Execute privilege.
- In an ANSI-compliant database, the creator of a UDR must explicitly grant the Execute privilege on the UDR for other users to be able to execute it.

In Informix®, if two or more UDRs have the same name, use a keyword from this list to specify which of those UDRs a user list can execute.

Keyword

UDR that the User Can Execute

SPECIFIC

The UDR identified by *specific name*

FUNCTION

Any function with the specified *routine name* (and parameter types that match *routine parameter list*, if specified)

PROCEDURE

Any procedure with the specified *routine name (*and parameter types that match *routine parameter list,* if specified)

ROUTINE

Functions or procedures with the specified *routine name (*and parameter types that match *routine parameter list,* if specified)

If both a user-defined function and a user-defined procedure of Informix® have the same name and the same list of parameter data types, you can grant the Execute privilege to both with the keyword ROUTINE.

To limit the Execute privilege to one routine among several that have the same identifier, use the FUNCTION, PROCEDURE, or SPECIFIC keyword.

To limit the Execute privilege to a UDR that accepts certain data types as arguments, include the routine parameter list or use the SPECIFIC keyword to introduce the specific name of a UDR.

If an external function has a negator function, you must grant the Execute privilege on both the external function and on its negator function before other users can execute the external function.

A user must hold the Usage privilege on the language in which the user-defined routine is written to register a UDR with the CREATE FUNCTION, CREATE FUNCTION FROM, CREATE PROCEDURE, CREATE PROCEDURE FROM, or CREATE ROUTINE FROM statement. For more information on the requirements to register a UDR, see Privileges necessary for using CREATE FUNCTION on page 228.

Granting the Execute privilege to PUBLIC

The GRANT statement supports syntax for granting access to the PUBLIC group, which includes all users who hold the Connect privilege on the database.

GRANT EXECUTE TO PUBLIC

```
GRANT Routine level privileges TOPUBLIC [ AS'grantor ' ]
```

Element	Description	Restrictions	Syntax
grantor	Owner of the UDR	Cannot be a role	Owner name on page 1262

This statement enables everyone in the PUBLIC group to execute the specified routine. It overrides the NODEFDAC environment variable, if that is set to prevent the PUBLIC group from receiving Execute privilege on the routine by default. This statement also enables users who do not hold the DBA privilege to execute the specified routine, whether that routine was created with the DBA keyword.

Only users who hold the DBA privilege can specify the AS *grantor* clause. The specified grantor must be the owner of the specified routine, as listed in the **owner** column of the **sysprocedures** system catalog table. The grantor cannot be the name of a role or the PUBLIC keyword.

In an ANSI-compliant database, the AS *grantor* clause is required, rather than optional, if the DBA who issues the GRANT EXECUTE statement is not the owner of the specified routine.

Only the user specified in the optional AS *grantor* clause can use the REVOKE statement of SQL to revoke the Execute privilege from the PUBLIC group

The user specified in the optional AS *grantor* clause can use the REVOKE statement to revoke the Execute privilege from the PUBLIC group.

Revoking the Execute privilege from PUBLIC

The REVOKE statement supports the following syntax for revoking access to the specified routine from the PUBLIC group, which includes all users who hold the Connect privilege on the database.

REVOKE EXECUTE TO PUBLIC

```
GRANT Routine level privileges FROMPUBLIC [ AS'revoker ' ]
```

Element	Description	Restrictions	Syntax
revoker	Owner of the UDR	Cannot be a role	Owner name on page 1262

This statement prevents the PUBLIC group from receiving Execute privileges on the specified routine by default. (Individual users who hold the DBA privilege, however, or who own the routine, or who were granted Execute privilege on the routine individually or through a role, are not affected by this statement.

Only users who hold the DBA privilege can specify the AS *revoker* clause. The specified revoker must be the owner of the specified routine, as listed in the **owner** column of the **sysprocedures** system catalog table. The name cannot be the name of a role or the PUBLIC keyword.

In an ANSI-compliant database, the AS *revoker* clause is required, rather than optional, if the DBA who issues the REVOKE EXECUTE statement is not the owner of the specified routine.

In databases where the PUBLIC group holds Execute privilege on owner-privileged routines by default, the REVOKE EXECUTE ON PUBLIC statement must be successfully executed before the discretionary access privilege to execute the specified routine can be granted to a subset of users or to one or more roles. Otherwise, only users with the DBA privilege or the owner of the routine can start it.

Language-Level Privileges

Informix® also supports *language-level privileges*, which specify the programming languages of UDRs that users who have been granted Usage privileges for a given language can register in the database.

This is the syntax of the USAGE ON LANGUAGE clause for granting Usage privileges on a programming language:

Language-Level Privileges

" USAGE ON LANGUAGE { SPL | C | JAVA } "

The SPL, C, and JAVA keywords can specify a programming language in the USAGE ON LANGUAGE clause. Each GRANT USAGE ON LANGUAGE statement can specify no more than one programming language. By default, Usage privilege on SPL is granted to PUBLIC.

When a user executes the CREATE FUNCTION or CREATE PROCEDURE statement to register a UDR that is written in the SPL, C, or Java™ language, the database server verifies that the user has the Usage privilege on the language in which the UDR is written. If the IFX_EXTEND_ROLE configuration parameter has enabled the built-in EXTEND role, only users who also hold that role can register or drop UDRs written in the C language or in the Java™ language, even if the users hold USAGE ON LANGUAGE privileges for those languages.

The GRANT USAGE ON LANGUAGE statement can grant Usage privilege on a programming language to a restricted group of users. The following example grants Usage privileges on the C language to a user-defined role named **developers**:

```
GRANT USAGE ON LANGUAGE C TO developers;
```

If the preceding example executes successfully, users who hold the **developers** role as their current role can create or drop C routines (if they also hold the EXTEND role, or if the IFX_EXTEND_ROLE parameter is set to `0` or to `Off`).

For information on other access privileges that these statements require, see CREATE FUNCTION statement on page 225 and CREATE PROCEDURE statement on page 270.

Usage Privilege in Stored Procedure Language

The Usage privilege on SPL is granted to PUBLIC by default. Only user **informix**, the DBA, or a user who was granted the Usage privilege WITH GRANT OPTION can grant the Usage privilege on SPL to another user.

In the following example, assume that the default Usage privilege on SPL was revoked from PUBLIC and the DBA wants to grant the Usage privilege on SPL to the role named **developers**:

```
GRANT USAGE ON LANGUAGE SPL TO developers;
```

Sequence-Level Privileges

Although Informix® implements sequence objects as tables, only a subset of table-level privileges (page Table-Level Privileges on page 540) can be granted on a sequence. You can grant the Select or Alter privilege (or both) on a sequence:

(explicit id unique_775_Connect_42_slp001) unique_775_Connect_42_slp001

Sequence-Level Privileges

" { ALL | { | SELECT | ALTER } } "

" ON "

" [*owner* .] "

" { *sequence* | *synonym* } "

Element	Description	Restrictions	Syntax
owner	Owner of sequence (or owner of *synonym*)	Must be the owner	Owner name on page 1262
sequence	Sequence on which to grant privileges	Must exist	Identifier on page 1231
synonym	Synonym for a sequence object	Must exist	Identifier on page 1231

The sequence object must exist in the current database. You can qualify the *sequence* or *synonym* identifier with a valid *owner* name, but the name of a remote *database* (or *database*@*server*) is not valid as a qualifier. You can include the WITH GRANT OPTION keywords when you grant ALTER, SELECT, or ALL to a user or to PUBLIC (but not to a role) as privileges on a sequence object.

Alter Privilege

You can grant the Alter privilege on a sequence to another user or to a role. The Alter privilege enables a specified user or role to modify the definition of a sequence with the ALTER SEQUENCE statement or to rename the sequence with the RENAME SEQUENCE statement.

The following statement grants the Alter privilege to user **mark** on the **cust_seq** sequence object:

```
GRANT ALTER ON cust_seq TO mark;
```

Select Privilege

You can grant the Select privilege on a sequence to another user or to a role. The Select privilege enables a specified user or role to use *sequence*.**CURRVAL** and *sequence*.**NEXTVAL** expressions in SQL statements to read and to increment (respectively) the value of a sequence.

The following statement grants the Select privilege to user **mark** on the **cust_seq** sequence object:

```
GRANT SELECT ON cust_seq TO mark;
```

ALL Keyword

You can specify the ALL keyword to grant both Alter and Select privileges on a sequence object to another user or to a role, or to a list of users or roles.

The User List

You can grant privileges to an individual user or to a list of users. You can also specify the PUBLIC keyword to grant privileges to all users.

User List

" { **PUBLIC** | { | *user* | *'user'* } } "

Element	Description	Restrictions	Syntax
user	Login name of a user to whom you are granting privilege or granting a role	Must be a valid authorization identifier	Owner name on page 1262

The following example grants the Insert table-level privilege on **table1** to the user **mary** in a database that is not ANSI-compliant:

```
GRANT INSERT ON table1 TO mary;
```

In an ANSI-compliant database, if you do not include quotation marks as delimiters around *user*, the name of the user is stored in uppercase letters.

Role Name

You can use the GRANT statement to associate a list of one or more users (or all users, using the PUBLIC keyword) with a *role* name that can describe what they do. After you declare and grant a role, access privileges that you grant to that role are thereby granted to all users who are currently associated with that role.

Role Name

" { *role* | *'role'* } "

Element	Description	Restrictions	Syntax
role	Role that is granted, or to which a privilege or another role is granted	Must exist. If enclosed between quotation marks, *role* is case sensitive.	Owner name on page 1262

You can also grant an existing role to another role. This action gives whatever privileges the granted role possesses to all users who have the receiving role.

Granting a Role to a User or to Another Role

You must register a role in the database before the role can be used in a GRANT statement. For more information, see CREATE ROLE statement on page 282.

A DBA has the authority to grant a new role to another user. If a user receives a role WITH GRANT OPTION, that user can grant the role to other users or to another role. Users keep a role that was granted to them until the REVOKE statement breaks the association between their login names and the role name.

Important: The CREATE ROLE and GRANT statements do not activate the role. A non-default role has no effect until SET ROLE enables it. The grantor or the grantee of a role can issue the SET ROLE statement.

The following example shows the actions required to grant and activate the role **payables** to a group of employees who perform account payable functions. First the DBA creates role **payables**, then grants it to **maryf**.

```
CREATE ROLE payables;
GRANT payables TO maryf WITH GRANT OPTION;
```

The DBA or **maryf** can activate the role with the following statement:

```
SET ROLE payables;
```

User **maryf** has the WITH GRANT OPTION authorization to grant **payables** to other employees who pay accounts.

```
GRANT payables TO charly, gene, marvin, raoul;
```

If you grant privileges for one role to another role, the recipient role has the combined set of privileges that have been granted to both roles. The following example grants the role **petty_cash** to the role **payables**:

```
CREATE ROLE petty_cash;
SET ROLE petty_cash;
GRANT petty_cash TO payables;
```

After all of these statements execute successfully, if user **raoul** uses the SET ROLE statement to make **payables** their current role, then (aside from the effects of any REVOKE operations) they (raoul) hold the following combined set of access privileges:

- The privileges granted to the **payables** role
- The privileges granted to the **petty_cash** role
- The privileges granted individually to **raoul**
- The privileges granted to PUBLIC

If you attempt to grant a role to yourself, either directly or indirectly, the database server generates an error. (For an important exception to this rule, however, see the description of the DBSECADM Clause on page 558.)

The database server also generates an error if you include the WITH GRANT OPTION keywords in a GRANT statement that assigns a role to another role.

Granting privileges to a role

You can grant table-level and routine-level access privileges to a role if you have the authority to grant these same privileges to login names or to PUBLIC. You can also grant type-level privileges to a role. A role cannot hold database-level privileges.

Important: The scope of a user-defined role (and of discretionary access privileges that the GRANT statement assigns to the role) is the current database. When the GRANT DEFAULT ROLE or SET ROLE statement activates a role, the role and its privileges take effect in the current database only. As a security precaution, discretionary access privileges that a user receives only from a role cannot provide access to tables outside the current database through a view or through the action of a trigger.

The syntax is more restricted for granting privileges to a role than to a user:

- You can specify the AS *grantor* clause.

  In this way, whoever has the role can revoke these same privileges. For more information, see AS grantor clause on page 557.

- You cannot include the WITH GRANT OPTION clause.

  A role cannot, in turn, grant the same access privileges to another user.

This example grants Insert privilege on the **supplier** table to the role **payables**:

```
GRANT INSERT ON supplier TO payables;
```

Anyone who has been granted the **payables** role, and who successfully activates it by issuing the SET ROLE statement, can now insert rows into **supplier**.

Granting a Default Role

The DBA or the owner of the database (by default, user **informix**) can define a *default role* for one or more users or for PUBLIC with the GRANT DEFAULT ROLE statement. A default role is activated when the user connects to the database. The SET ROLE statement is not required to activate a default role.

Default roles are useful if users access databases through client applications that cannot modify access privileges nor set roles.

A default role can specify a set of access privileges to all the users who are assigned that role, as in the following example:

```
CREATE ROLE accounting;
GRANT ALTER, INSERT, SELECT ON stock TO accounting;
GRANT DEFAULT ROLE accounting TO mary, asok, vlad;
```

The last statement provides users **mary**, **asok**, and **vlad** with **accounting** as their default role. If any of these users connects to a database, that user activates whatever privileges the **accounting** role holds, in addition to any privileges that the user already possesses as an individual or as PUBLIC.

The role must already exist and the user must have the access privileges to set the role. If the role has not previously been granted to a user, it is granted as part of setting the default role.

If no default role is defined for a user nor for PUBLIC, then no role is set, and the existing privileges of the user are in effect.

The following example shows how the default role can be assigned to all users:

```
DATABASE hrdb;
CREATE ROLE emprole;
GRANT CONNECT TO PUBLIC;
GRANT SELECT ON emptab TO emprole;
GRANT emprole TO PUBLIC;
GRANT DEFAULT ROLE emprole TO PUBLIC;
```

Note: Using GRANT DEFAULT ROLE is an alternative to issuing the SET ROLE statement in the **sysdbopen( )** procedure. Default roles defined using the **sysdbopen( )** procedure, however, have precedence over any other role when a user establishes a connection.

Changing the default role for a user or for PUBLIC only affects new database connections. Existing connection continue to run under currently assigned roles. If one default role was granted to *user*, and another default role was granted to PUBLIC, the default role granted to *user* takes precedence at connection time.

A default role cannot be assigned to another role. Because roles are not defined across databases, the default role must be assigned for each database. No options besides the *user-list* are valid after the TO keyword in the GRANT DEFAULT ROLE statement. The database server issues an error if you attempt to include the AS *grantor* clause or the WITH GRANT OPTION clause.

Granting the EXTEND Role

If the IFX_EXTEND_ROLE configuration parameter is set to `ON` or to `1`, only users who hold the EXTEND role (and who also hold the Resource privilege on the database and the Usage privilege on the programming language in which the UDR is written) can create or drop UDRs that are written in the C or Java™ external languages that can support shared libraries.

The Database Server Administrator (DBSA), by default user **informix**, can grant the EXTEND role to one or more users or to PUBLIC with the GRANT EXTEND TO *user-list* statement.

Because EXTEND is a built-in role, the SET ROLE statement is not required for the EXTEND role to have this effect. It is sufficient for a user to hold the EXTEND role without using SET ROLE to enable it.

For example, suppose that user **max** holds Resource privileges on the database, and has also been granted Usage privilege on the C language by the GRANT USAGE ON LANGUAGE C statement. The following statement grants the EXTEND role to user **max**:

```
GRANT EXTEND TO 'max';
```

This statement enables user **max** to create or drop UDRs that are written in the C language, without requiring **max** to issue the SET ROLE EXTEND statement. (Here the quotation marks preserve the lowercase letters in the authorization identifier **max**.) Before user **max** can create or drop UDRs written in the Java™ language, however, the TO clause of a valid GRANT

USAGE ON LANGUAGE JAVA statement must specify either `'max'`, or `PUBLIC`, or the name of a user-defined role that **max** holds (and that **max** has used the SET ROLE statement to specify as their current role).

In databases for which this security feature is not needed, the DBSA can disable this restriction on who can create or drop external UDRs by setting the IFX_EXTEND_ROLE configuration parameter to `OFF` or to `0` in the ONCONFIG file. When IFX_EXTEND_ROLE is set to `OFF` or to `0`, any user who holds the Resource privilege (and also holds the Usage privilege on the programming language in which the UDR is written) can create or drop external UDRs.

Resource privileges on the database and Usage privilege on the external language are required for any user to create or drop an external UDR, regardless of the IFX_EXTEND_ROLE configuration parameter setting, or whether the user holds the EXTEND role. User **informix**, the DBA, or any user who has received Usage privileges WITH GRANT OPTION can grant Usage privileges on the SPL, C, and Java™ languages to PUBLIC. See Database-Level Privileges on page 539 for information about granting the Resource privilege. See Language-Level Privileges on page 549 for information about granting Usage privileges on programming languages.

WITH GRANT OPTION keywords

The WITH GRANT OPTION keywords convey the privilege or role to a *user* with the right to grant the same privileges or role to other users.

You create a chain of privileges that begins with you and extends to *user* as well as to whomever *user* subsequently conveys the right to grant privileges. If you include WITH GRANT OPTION, you can no longer control the dissemination of privileges.

The following example grants the Alter and Select privileges to user **mark** on the **cust_seq** sequence object, with the ability to grant those privileges to others:

```
GRANT ALL ON cust_seq TO mark WITH GRANT OPTION;
```

If you revoke from *user* the privilege that you granted using the WITH GRANT OPTION keyword, you sever the chain of privileges. That is, when you revoke privileges from *user*, you automatically revoke the privileges of all users who received privileges from *user* or from the chain that *user* created (unless *user*, or the users who received privileges from *user*, were granted the same set of privileges by someone else).

The following examples illustrate this situation. You, as the owner of the table **items**, issue the following statements to grant access to user **mary**:

```
REVOKE ALL ON items FROM PUBLIC;
GRANT SELECT, UPDATE ON items TO mary WITH GRANT OPTION;
```

User **mary** uses their privilege to grant users **cathy** and **paul** access to the table:

```
GRANT SELECT, UPDATE ON items TO cathy;
GRANT SELECT ON items TO paul;
```

Later you revoke the access privileges for user **mary** on the **items** table:

```
REVOKE SELECT, UPDATE ON items FROM mary;
```

This single statement effectively revokes all privileges on the **items** table from users **mary**, **cathy**, and **paul**. If you want to create a chain of privileges with another user as the source of the privilege, use the AS *grantor* clause.

In Informix®, the WITH GRANT OPTION keywords are valid only for users. They are not valid when a role is the grantee of a privilege or of another role. You cannot specify WITH GRANT OPTION in a statement that grants a privilege to the PUBLIC group.

The Database Server Administrator cannot include the WITH GRANT OPTION keywords in the GRANT EXTEND or GRANT DBSECADM statements. The DBSA cannot delegate to another user the authorization to grant the built-in EXTEND or DBSECADM roles. If more than one user needs either of these authorizations, they should be included in the **DBSA** group when the database server is installed.

In addition to the GRANT DBSECADM statement, none of the other security administration options of the GRANT statement support the WITH GRANT OPTION keywords. For more information about these statements and their syntax, see Security Administration Options on page 558.

AS *grantor* clause

When you grant discretionary access privileges to other users, roles, or to PUBLIC, by default you are the one who can revoke those privileges. The AS *grantor* clause lets you establish another user as the source of the privileges that you are granting.

When you use the AS *grantor* clause, the login provided in the AS *grantor* clause replaces your login in the appropriate system catalog table. You can use this clause only if you have the DBA privilege on the database.

After you use this clause, only the specified *grantor* can revoke the effects of the current GRANT operation. Even a DBA cannot revoke a privilege unless that DBA is listed in the system catalog table as the user who granted the privilege.

The following example illustrates this situation. You are the DBA and you grant all privileges on the **items** table to user **tom** with the right to grant all privileges:

```
REVOKE ALL ON items FROM PUBLIC;
GRANT ALL ON items TO tom WITH GRANT OPTION;
```

The next example illustrates a different situation. You also grant Select and Update privileges to user **jim**, but you specify that the privileges are granted as user **tom**. (The records of the database server in the **systabauth** system catalog table show that user **tom** is the grantor of those privileges, rather than you.)

```
GRANT SELECT, UPDATE ON items TO jim AS tom;
```

Later, you decide to revoke privileges on the **items** table from user **tom**, so you issue the following statement:

```
REVOKE ALL ON items FROM tom;
```

If instead, however, you try to revoke privileges from user **jim** with a similar statement, the database server returns an error, as the next example shows:

```
REVOKE SELECT, UPDATE ON items FROM jim;

580: Cannot revoke permission.
```

You receive an error because the database server record shows the original grantor as user **tom**, and you cannot revoke the privilege. Although you are the DBA, you cannot revoke a privilege that another user granted.

The AS *grantor* clause is not valid in the GRANT DEFAULT ROLE statement.

For contexts where the AS *grantor* clause is required, rather than optional, see Granting the Execute privilege to PUBLIC on page 548.

Security Administration Options

In conjunction with the REVOKE statement, the GRANT statement supports the discretionary access control (DAC) data security feature of Informix® by specifying which users or roles hold privileges that are required to access the database or objects within the database.

The Security Administration Options of the GRANT statement, like their counterparts for the REVOKE statement, support an additional set of data security features, called label-based access control (LBAC). These features enable Informix® to allow or withhold access to protected data on the basis of a comparing a row security label or column security label that is contained in the data object to the user security label and other credentials that have been granted to the user who is seeking access.

(explicit id unique_777_Connect_42_sao002) unique_777_Connect_42_sao002(explicit id unique_777_Connect_42_sao003)

unique_777_Connect_42_sao003(explicit id unique_777_Connect_42_sao004) unique_777_Connect_42_sao004(explicit id unique_777_Connect_42_sao005)

unique_777_Connect_42_sao005(explicit id unique_777_Connect_42_sao006) unique_777_Connect_42_sao006

Security Administration Options

```
" { <DBSECADM Clause> [] | <EXEMPTION Clause> [] | <SECURITY LABEL Clause> [] | <SETSESSIONAUTH Clause> [] } "
```

Use of these GRANT statement security administration options is restricted:

- Only the Database Server Administrator (DBSA), by default user **informix**, or (on UNIX™) a member of the **DBSA** group, or (on Windows™) a member of the **Informix-Admin** group, can use the GRANT DBSECADM statement to grant the DBSECADM role.
- Only a user who holds the DBSECADM role can issue the GRANT EXEMPTION, GRANT SECURITY LABEL, or GRANT SETSESSIONAUTH statements, or the corresponding REVOKE statements.

DBSECADM Clause

The GRANT DBSECADM statement enables the user to whom the DBSECADM role is granted to issue DDL statements that can create, alter, rename, or drop security objects, including security policies, security labels, and security components.

DBSECADM Clause

" DBSECADM TO "

" [USER] *user* "

Element	Description	Restrictions	Syntax
user	User to whom the role is to be granted	Must be the authorization identifier of a user	Owner name on page 1262

The DBSECADM role is a built-in role that only the DBSA can grant. Unlike user-defined roles, whose scope is the database in which the role is created, the scope of the DBSECADM role is all of the databases of the Informix® instance. It is not necessary for DBSA to reissue the GRANT DBSECADM statement in other databases of the same server. Like all built-in roles of Informix®, the DBSECADM role is enabled when it is granted, without requiring activation by the SET ROLE statement, and it remains in effect until it is revoked.

Only a user who holds the DBSECADM role can issue the following SQL statements that create or modify security objects:

- ALTER SECURITY LABEL COMPONENT
- CREATE SECURITY LABEL
- CREATE SECURITY LABEL COMPONENT
- CREATE SECURITY POLICY
- DROP SECURITY LABEL
- DROP SECURITY LABEL COMPONENT
- DROP SECURITY POLICY
- RENAME SECURITY LABEL
- RENAME SECURITY LABEL COMPONENT
- RENAME SECURITY POLICY

Only a user who holds the DBSECADM role can use the following SQL statements to reference tables that are protected by a security policy:

- ALTER TABLE ... ADD SECURITY POLICY
- ALTER TABLE ... ADD ... IDSSECURITYLABEL [DEFAULT *label*]
- ALTER TABLE ... ADD ... [COLUMN] SECURED WITH
- ALTER TABLE ... DROP SECURITY POLICY
- ALTER TABLE ... MODIFY ... [COLUMN] SECURED WITH
- ALTER TABLE ... MODIFY ... DROP COLUMN SECURITY
- CREATE TABLE ... COLUMN SECURED WITH

- CREATE TABLE ... IDSSECURITYLABEL [DEFAULT *label*]
- CREATE TABLE ... SECURITY POLICY

The following GRANT and REVOKE statements also cannot be issued by a user who does not hold the DBSECADM role:

- GRANT EXEMPTION
- GRANT SECURITY LABEL
- GRANT SETSESSIONAUTH
- REVOKE EXEMPTION
- REVOKE SECURITY LABEL
- REVOKE SETSESSIONAUTH

The USER keyword that can follow the TO keyword is optional, and has no effect, but any authorization identifier that the DBSA specifies in the GRANT DBSECADM statement must be the identifier of an individual user, rather than the identifier of a role, or the PUBLIC group.

The *user* can be the DBSA who issues this GRANT DBSECADM statement. This is an important exception to the general restriction that the TO clause of the GRANT statement (like the FROM clause in the REVOKE statements) cannot explicitly reference the authorization identifier of the user who issues the statement. Unlike other roles, access privileges, user security labels, and exemptions on rules that the GRANT statement can specify, you can grant the DBSECADM role to yourself, if you are user **informix**, or a member of the DBSA group, or (on Windows™ if you are a member of the **Informix-Admin** group.

In the following example, the DBSA grants the DBSECADM role to user **niccolo**:

```
GRANT DBSECADM TO niccolo;
```

If this statement executes successfully, user **niccolo** can perform the LBAC operations listed above, provided that **niccolo** also holds sufficient discretionary access privileges on the database and on the database objects that those SQL statements reference.

After a user is granted the DBSECADM role, only the DBSA can revoke it.

For a discussion of LBAC security objects, see your HCL Informix® Security Guide.

EXEMPTION Clause

The GRANT EXEMPTION statement modifies the security credentials of the specified user (or list of users) by disabling one or all of the rules of a specified security policy.

EXEMPTION Clause

"EXEMPTION ON RULE"

"{ IDSLBACREADARRAY | IDSLBACREADTREE | IDSLBACREADSET | IDSLBACWRITEARRAY [{ WRITEDOWN | WRITEUP }] | IDSLBACWRITESET | IDSLBACWRITETREE | ALL }"

"FOR *policy*"

"TO [USER] *user*"

Element	Description	Restrictions	Syntax
policy	The security policy from which the exemption is granted	Must exist in the database	Identifier on page 1231
user	User to whom the exemption is to be granted	Must be the authorization identifier of a user	Owner name on page 1262

Only a user who holds the DBSECADM role can issue the GRANT EXEMPTION statement.

Rules on Which Exemptions Are Granted

The keyword that follows the ON keyword specifies the predefined LBAC access rule of the security policy (whose identifier follows the FOR keyword) for which an exemption is granted. The access rule for which exemption is granted does not apply when a table that is protected by the specified security policy is accessed by a user to whom the exemption is granted. For descriptions of the predefined rules for read access and for write access that are associated with a security policy, see the section Rules Associated with a Security Policy on page 301.

The following keywords of the GRANT EXEMPTION statement identify specific **IDSLBACRULES** rules from which this statement can exempt users:

- IDSLBACREADARRAY exempts the user from the **IDSLBACREADARRAY** rule for the specified security policy. That rule requires that each array component of the user security label must be greater than or equal to the corresponding array component of the data row security label.
- IDSLBACREADSET exempts the user from the **IDSLBACREADSET** rule for the specified security policy. That rule requires that each set component of the user security label must include the set component of the data row security label
- IDSLBACREADTREE exempts the user from the **IDSLBACREADTREE** rule for the specified security policy. That rule requires that each tree component of the user security label must include at least one of the elements in the tree component of the data row security label, or else the ancestor of one such element.

- IDSLBACWRITEARRAY WRITEDOWN exempts the user from one aspect of the **IDSLBACWRITEARRAY** rule for the specified security policy. That rule requires that each array component of the user security label must be equal to the array component of the data row security label. The user who holds this exemption can write to a row whose array component level is below the level in the label of the user. The user cannot, however, write to a row in whose label the array component level is above the level in the label of the user.
- IDSLBACWRITEARRAY WRITEUP exempts the user from one aspect of the **IDSLBACWRITEARRAY** rule for the specified security policy. The user who holds this exemption can write to a row whose array component level is above the level in the label of the user. The user cannot, however, write to a row in whose label the array component level is below the level in the label of the user.
- IDSLBACWRITEARRAY (with no WRITEDOWN or WRITEUP keyword) exempts the user from the **IDSLBACWRITEARRAY** rule for the specified security policy. The user who holds this exemption can write to a row without regard to the corresponding array component level of the row label.
- IDSLBACWRITESET exempts the user from the **IDSLBACWRITESET** rule for the specified security policy. That rule requires that each set component of the user security label must include the set component of the data row security label
- IDSLBACWRITETREE exempts the user from the **IDSLBACWRITETREE** rule for the specified security policy. That rule requires that each tree component of the user security label must include at least one of the elements in the tree component of the data row security label, or an ancestor of one such element.
- ALL exempts the user from all **IDSLBACRULES** rules for the specified security policy. This form of exemption is required to load data into a protected table.

In the following example, DBSECADM grants an exemption from all of the rules of the **MegaCorp** security policy to users **manoj** and **sam**:

```
GRANT EXEMPTION ON RULE ALL FOR MegaCorp TO manoj, sam;
```

Security Policies and Grantees of Exemptions

An exemption applies only to the rules of the security policy whose name follows the FOR keyword. A protected table can have multiple security labels, but no more than one security policy.

The GRANT EXEMPTION statement fails with an error if the specified policy does not exist in the database.

The USER keyword that can follow the TO keyword is optional, and has no effect, but any authorization identifier specified in the GRANT EXEMPTION statement must be the identifier of an individual user, rather than the identifier of a role. This *user* cannot be the DBSECADM who issues the same GRANT EXEMPTION statement.

In the following example, DBSECADM grants an exemption to user **lynette** from rule **IDSLBACREADARRAY** of the **MegaCorp** security policy:

```
GRANT EXEMPTION ON RULE IDSLBACREADARRAY FOR MegaCorp TO lynette;
```

This exemption bypasses the read access rules for all array components of security labels of the specified policy.

When the GRANT EXEMPTION statement successfully grants an exemption to a user, the database server updates the **syssecpolicyexemptions** table of the system catalog to register the new exemption (or multiple exemptions, if several users are listed after the TO keyword).

For a discussion of LBAC security objects, see your Informix®.

SECURITY LABEL Clause

The GRANT SECURITY LABEL statement grants a security label to a user or to a list of users.

SECURITY LABEL Clause

" SECURITY LABEL *policy* . *label* "

" TO [USER] *user* "

" { FOR ALL ACCESS | [FOR READ ACCESS] | [FOR WRITE ACCESS] } "

Element	Description	Restrictions	Syntax
label	Name of an existing security label	Must exist as a label for the specified security *policy*	Identifier on page 1231
policy	The security policy of this *label*	Must already exist in the database	Identifier on page 1231
user	User to whom the label is to be granted	Must be the authorization identifier of a user	Owner name on page 1262

Only a user who holds the DBSECADM role can issue the GRANT SECURITY LABEL statement.

A security label is a database object that is always associated with a security policy. That policy defines the set of valid security components that make up the security label. The label stores a set of one or more values for each component of the security policy.

The DBSECADM can associate a security label with the following entities:

- A column of a database table, which a *column security label* can protect
- A row of a database table, which a *row security label* can protect
- A user, whose *user security label* (and any exemptions from rules of the security policy that have been granted to the user) are called the *security credentials* of the user.

When a user who holds a security label for a specific security policy attempts to access a row that is protected by a row security label of the same security policy, the database server compares the sets of values in the user security label and in

the row security label to determine whether the user should be allowed access to the data. Similarly, LBAC takes into account the user security label and the column security label in determining whether the credentials of the user should allow access to the protected column.

The GRANT SECURITY LABEL statement is the mechanism by which DBSECADM associates a user with a security label. (Data values in a protected table are associated with a row security label or with a column security label by options to the CREATE TABLE or ALTER TABLE statements that only DBSECADM can execute, rather than by the GRANT SECURITY LABEL statement.)

The USER keyword that can follow the TO keyword is optional, and has no effect, but any authorization identifier specified in the GRANT SECURITY LABEL statement must be the identifier of an individual user, rather than of a role.

Access Specifications

The list of users to whom the security label is granted can optionally be followed by keywords that specify the type of access to data that the security policy of the label protects

- FOR WRITE ACCESS

  These keywords restrict the label to the write access rules of **IDSLBACRULES**, namely **IDLSBACWRITEARRAY**, **IDLSBACWRITESET**, and **IDLSBACWRITETREE**. These rules affect INSERT, DELETE, and UPDATE operations on protected data.

- FOR READ ACCESS

  These keywords restrict the label to the read access rules of **IDSLBACRULES**, namely **IDLSBACWREADARRAY**, **IDLSBACREADSET**, and **IDLSBACREADTREE**. These rules affect SELECT, DELETE, and UPDATE operations on protected data.

- FOR ALL ACCESS

  These keywords apply the label to all of the read and write access rules that are listed above. If the GRANT SECURITY LABEL statement includes no FOR ... ACCESS specification, this option takes effect as the default.

For more information about these **IDSLBACRULES** rules for label-based read and write access, see Rules Associated with a Security Policy on page 301. For information about exemptions to these rules that can be granted for a specific security policy, see Rules on Which Exemptions Are Granted on page 561.

If a user is granted a different security label for read access than for write access, then the values given for the security label components must follow these rules:

- For security label components of type ARRAY, the value must be the same in both security labels.
- For security label components of type SET, the values given in the security label used for WRITE access must be a subset of the values given in the security label used for READ access. If all of the values are the same, this is interpreted as being a subset, and is allowed.
- For security label components of type TREE, every element in the tree component of the security label for write access must be either an element or a descendent of an element in the tree component of the security label for read access.

In summary, when DBSECADM attempts to grant a security label for read access to a user who already holds a security label for write access, or vice versa, the read label cannot be more restrictive than the write label. Otherwise, the GRANT SECURITY LABEL statement fails with an error.

A user can be granted no more than two labels for the same security policy. If two labels are granted for the same policy, one label must be for read access and the other for write access. If DBSECADM attempts to grant a security label for read access to a user who already holds a security label for read access that is based on the same security policy, the GRANT SECURITY LABEL statement fails with an error. A similar failure result if both labels are for write access and are on the same security policy.

In both of these cases, the first security label must be revoked explicitly by the REVOKE SECURITY LABEL statement before a second label can be granted for the same access mode and the same security policy. The only exception to this rule is if both labels specify the same values for component elements, because in this case both security labels are functionally identical, and no error is issued.

Rules for User Security Labels

The following rules affect security labels that are granted to users by the GRANT SECURITY LABEL statement:

- The *user* cannot be the DBSECADM who issues this GRANT SECURITY LABEL statement.
- A user without a security label has a NULL or zero label. A user with no security label cannot access data in a protected table, unless the user holds the necessary exemptions on the policy.
- By default, the IDSSECURITYLABEL column of a protected table cannot have NULL values. A user with no security label cannot insert data into a table with row protection, even if the user has been granted the necessary exemptions on the security policy, unless the row label is explicitly specified in the INSERT statement. For details of how to specify a security label explicitly in the INSERT statement, see Security Label Support Functions on page 1092.
- User security labels have no effect on the following types of database tables, because these tables cannot be protected by a security policy:
  - Virtual Table Interface tables,
  - tables with Virtual Index Interface indexes,
  - tables in a typed-table hierarchy,
  - temporary tables.

Examples of Granting User Security Labels

The following three statements create three security label components called **level**, **compartments**, and **groups** respectively:

```
CREATE SECURITY LABEL COMPONENT
   level ARRAY ['TS','S','C','U'];

CREATE SECURITY LABEL COMPONENT
   compartments SET {'A','B','C','D'};

CREATE SECURITY LABEL COMPONENT
   groups TREE ('G1' ROOT,
                'G2' UNDER ROOT,
                'G3' UNDER ROOT);
```

The following statement creates a security policy called **secPolicy** based on the three components above:

```
CREATE SECURITY POLICY secPolicy COMPONENTS
   level, compartments, groups;
```

The following statement creates a security label called **secLabel1**:

```
CREATE SECURITY LABEL secPolicy.secLabel1
   COMPONENT level 'S',
   COMPONENT compartments 'A', 'B',
   COMPONENT groups 'G2';
```

The following statement creates a security label called **secLabel2**:

```
CREATE SECURITY LABEL secPolicy.secLabel2
   COMPONENT level 'S',
   COMPONENT compartments 'B',
   COMPONENT groups 'G2';
```

The following statement creates a security label called **secLabel3**:

```
CREATE SECURITY LABEL secPolicy.secLabel3
   COMPONENT level 'S',
   COMPONENT compartments 'A',
   COMPONENT groups 'G3';
```

The following statement creates a security label called **secLabel4**:

```
CREATE SECURITY LABEL secPolicy.secLabel4
   COMPONENT level 'TS',
   COMPONENT compartments 'A',
   COMPONENT groups 'G1';
```

The following statement grants a security label for read access to user **sam**:

```
 GRANT SECURITY LABEL secPolicy.secLabel1
   TO sam FOR READ ACCESS;
```

The following statement grants a security label for write access to user **sam**. This statement succeeds because it satisfies the rules given above.

```
 GRANT SECURITY LABEL secPolicy.secLabel2
   TO sam FOR WRITE ACCESS;
```

The following statement grants a security label for read access to user **lynette**:

```
GRANT SECURITY LABEL secPolicy.secLabel1
   TO lynette FOR READ ACCESS;
```

The following statement attempts to grant a security label for write access to user **sam**. This statement fails because it violates the rule with respect to the tree component.

```
GRANT SECURITY LABEL secPolicy.secLabel3
   TO sam FOR WRITE ACCESS;
```

The following statement attempts to grant a security label for write access to user **sam**. This statement fails because it violates the rule with respect to the array component.

```
 GRANT SECURITY LABEL secPolicy.secLabel4
   TO sam FOR WRITE ACCESS;
```

When the GRANT SECURITY LABEL statement successfully grants a security label to a user, the database server updates the **sysseclabelauth** table of the system catalog to register the new holder of the security label.

For a discussion of LBAC security objects, see your HCL Informix® Security Guide

SETSESSIONAUTH Clause

The GRANT SETSESSIONAUTH statement grants the SETSESSIONAUTH privilege to one or more users or roles. This privilege allows the holder to use the SET SESSION AUTHORIZATION statement to set the session authorization to PUBLIC or to any one of a list of specified users.

SETSESSIONAUTH Clause

" **SETSESSIONAUTH ON** { **PUBLIC** | [**USER**] *user* } "

" **TO** { | [**USER**] *user* | [**ROLE**] *role* } "

Element	Description	Restrictions	Syntax
role	Role to which the privilege is to be granted	Must be the authorization identifier of a role	Owner name on page 1262
user	After the TO keyword, a user to whom the privilege is to be granted. After the ON keyword, a user whose identity the grantee can specify in the SET AUTHORIZATION statement.	Must be the authorization identifier of a user	Owner name on page 1262

Only a user who holds the DBSECADM role can grant the SETSESSIONAUTH privilege. Both the SETSESSIONAUTH privilege and the DBA privilege are required to execute the SET AUTHORIZATION statement.

The user or PUBLIC specification that follows the ON keyword specifies whose identity the grantee of the SETSESSIONAUTH privilege can take while using SET SESSION AUTHORIZATION statement. This can be a user or PUBLIC but not a role. If PUBLIC is specified, then the grantee of the privilege can assume the identity of any database user.

The USER and ROLE keywords that can follow the TO keyword are optional. Neither the *user* nor the *role* can be the holder of the DBSECADM role who issues the GRANT SETSESSIONAUTH statement. The TO clause cannot specify PUBLIC as the grantee.

The following example grants to user **sam** the ability to set the session authorization to users **lynette** and **manoj**:

```
GRANT SETSESSIONAUTH ON lynette, manoj TO sam;
```

The next example grants to user **lynette** the ability to set the session authorization to PUBLIC:

```
GRANT SETSESSIONAUTH ON PUBLIC TO lynette;
```

Only a user who holds the DBSECADM role can revoke the SETSESSIONAUTH privilege. For a discussion of LBAC security objects, see your HCL Informix® Security Guide

Surrogate user properties (UNIX™, Linux™)

Use the ACCESS TO PROPERTIES clause of the GRANT statement to map users to surrogate user properties required for access to Informix® resources.

Note:

Only a DBSA can map externally authenticated users to valid surrogate user properties. If the USERMAPPING configuration parameter is set to enable support for mapped users, a DBSA issues the GRANT ACCESS TO PROPERTIES statement to map users to properties that correspond to the appropriate level of authorization.

Mapped users can connect to Informix® with the surrogate user properties if they authenticate with a pluggable authentication module (PAM) or single sign-on (SSO).

(explicit id unique_774_Connect_42_gract001) unique_774_Connect_42_gract001

ACCESS TO PROPERTIES Clause

" **ACCESS TO** { { **PUBLIC** | { | *user* } } } **PROPERTIES** "

" { **UID***user_ID*, **GROUP**({ | *group_ID* | *group_name* }) | **USER***OS_user_name* [, **GROUP**({ | *group_ID* | *group_name* })] } "

" [**HOME** "*directory* "] "

" [, **AUTHORIZATION** ({ | *privilege* })] "

Element	Description	Restrictions	Syntax
directory	Path name of directory where user files are stored.	Length cannot exceed 255 bytes, and must conform to the rules of your operating system. The *directory* must also: • Belong to the mapped *user_ID* and *group_ID* • Have read, write, and execute permissions for the owner • Not have PUBLIC write permissions	Quoted String on page 1202
group_ID	Group identifier number to which you want to map *user*. The list of *group_id* value or values that you specify must be enclosed in parentheses. .	The *group_ID* cannot be: • A group ID with server administrative privileges (DBSA, DBSSO, AAO, and BARGROUP) • Group 0 (**root**, sometimes referred to as **wheel** or **system**) • Group 80 on Mac OS X (**admin**) • A group ID associated with group **bin** or group **sys** The group ID must be present in `/etc/informix/allowed.surrogates` file.	Literal Number on page 1198
group_name	Name of an existing operating system group having the permissions to which you want to map *user* . The list of *group_name*	Length cannot exceed 32 bytes. The group name must be present in	Owner name on page 1262

Element	Description	Restrictions	Syntax
	values must be enclosed in parentheses.	/etc/informix/allowed.surrogates file.	
privilege	Administrative privilege to assign *user*. Valid values are as follows: • DBSA • DBSSO • AAO • BARGROUP The *privilege* value or values must be enclosed in parentheses.	The **USERMAPPING** configuration parameter must be set to ADMIN to grant server administrative privileges with the AUTHORIZATION keyword.	Quoted String on page 1202
user	Authorization identifier of a specific user that you are mapping to user properties.	Must be an authenticated authorization identifier	Owner name on page 1262
user_ID	User identifier number to which you want to map *user*.	*user_ID* cannot be the one that belongs to user **root** or user **informix**. The user ID must be present in /etc/informix/allowed.surrogates file.	Literal Number on page 1198
OS_user_name	Name of an existing OS user account on the Informix® host computer having the permissions to which you want to map *user*.	Must conform to the rules of your operating system . The user name must be present in /etc/informix/allowed.surrogates file.	Owner name on page 1262

Usage

The best practice is to map *user* to a specific OS user name that is reserved as a surrogate user identity only. You can add groups associated with the surrogate user identity with the GROUP keyword, and change the home directory with the HOME keyword. If the operating system administrator has specified acceptable surrogates in the `/etc/informix/allowed.surrogates` file, you can only map users to those specified OS users or groups.

If you map *user* to a user ID number, then remember to not create a user account on the Informix® host computer with the same number.

The USERMAPPING configuration parameter must be set to ADMIN in order to assign *user* a server administrative privilege with the ADMINISTRATOR keyword.

Note:

Use of this AUTHORIZATION clause (and of the AUTHORIZATION clause of the ALTER USER, CREATE USER, or CREATE DEFAULT USER statements) is not recommended. Different syntax will support role separation in a future release.

The PUBLIC and AUTHORIZATION keywords cannot be used together in the same statement, because the PUBLIC group cannot be granted server administrator privileges.

Specifying a directory for the user files with the HOME keyword is optional, but in some cases it is highly recommended. When an externally authenticated user is mapped to a surrogate user name but no HOME directory is specified in the GRANT ACCESS TO statement, then the mapped user has the same home directory as the user account on the Informix® host computer. When a user is mapped to a surrogate user identity with no set home directory, then Informix® creates a directory for user files in `$INFORMIXDIR/users`. In the latter case, the directory name in `$INFORMIXDIR/users` takes the form `uid.`*ID_number* (for example, `uid.101`).

Examples

The syntax and explanations in this section are examples for the following environment, where the acronym GID abbreviates *group ID number*, and the acronym UID abbreviates *user ID number*:

- There is a user **fred** with an OS account on the Informix® host computer. User **fred** has database server access with UID 3000, GID 3000 (**users**), auxiliary group 200 (**staff**), and home directory `/home/fred`.
- On the same computer, there exists an OS account for user **dbuser**. This account is locked so that **dbuser** cannot log in. The **dbuser** account exists only for the purpose of surrogate user mapping. It has UID 3050, GID 4000 (**ifx_user**), and home directory `/home/dbuser`.
- The group **ifx_user** has GID 4000, with users **bill** and **eileen**.
- The administrator setting up mapped users knows that there is no entry for UID 101 in `/etc/passwd` (or its equivalent) and no entry for GID 10011 or 10101 in `/etc/group` (or its equivalent) .
- User **bob** does not have OS account on the Informix® host computer but can authenticate through PAM or LDAP. The database server is configured to accept authentication through the PAM or LDAP module.
- The USERMAPPING parameter in the `onconfig` file is set to ADMIN.

Mapping an externally authenticated user to a surrogate user name:

The administrator maps **bob** to the database server access privileges that already exist for user **fred** by issuing the following GRANT statement:

```
GRANT ACCESS TO bob PROPERTIES USER fred;
```

Granting Informix® access to all externally authenticated users:

In this environment, the purpose of the user **dbuser** account on the Informix® host computer is to grant database server access to mapped users. In a situation where there are many mapped users and they do not need to know about the user files created in the home directory, the administrator might find it efficient and sufficiently secure to map PUBLIC to the **dbuser** surrogate user identity. The administrator can map all authenticated users (PUBLIC) to the privileges established for **dbuser** with the following GRANT ACCESS statement:

```
GRANT ACCESS TO PUBLIC PROPERTIES USER dbuser;
```

Note: The mapping of PUBLIC to a surrogate user identity is designed for mapped users who do not create or are not concerned with user files on the home directory, such as a consumer who is accessing Informix® databases on a retail Web site. If you want to map users concerned with the functioning of the database server to a surrogate user identity like **dbuser**, it is recommended to map these users individually with a designated home directory, as in the following example:

```
GRANT ACCESS TO bob PROPERTIES USER dbuser HOME "/home/dbuser/bob";
```

Mapping an externally authenticated user to a UID-GID pair:

The administrator maps **bob** to a surrogate user identity that consists of a UID-GID pair that enables database server access by running the following statement:

```
GRANT ACCESS TO bob PROPERTIES UID 101, GROUP (10011);
```

Because no specific directory was specified, a directory under `$INFORMIXDIR/users` will be created with the name `uid.101` and this path will be used as the home directory. The UID 101 and GROUP (10011) are anonymous because they do not have entries in the respective `/etc` directories that designate UIDs and GIDs that can access Informix® .

Alternatively, the administrator can map **bob** to a surrogate user identity that is a combination of an anonymous UID and to an explicit group, such as in the following example:

```
GRANT ACCESS TO bob PROPERTIES 101, GROUP (ifx_user);
```

Because the **ifx_user** group has members **bill** and **eileen**, the group is not anonymous.

Mapping an externally authenticated user to a surrogate user identity that has server administrative privileges:

In the following example, the administrator grants DBSA privileges to **bob**:

```
GRANT ACCESS TO bob PROPERTIES USER fred, GROUP (ifx_user), AUTHORIZATION (dbsa);
```

User **bob** is assigned UID 3000 (**fred**) and GIDs 3000 (**users**), 200 (**staff**), and the extra group 1000 (**ifx_user**). The administrative role granted to **bob** could be different by replacing `dbsa` with a different privilege (DBSSO, AAO, or BARGROUP). If the USERMAPPING parameter were set to BASIC in the `onconfig` file, then **bob** would not be granted DBSA privileges by this statement. If USERMAPPING were set to OFF, then **bob** would not be able to connect to the database server at all.

Examples

The following GRANT statements are examples of valid ACCESS TO PROPERTIES clauses, using hypothetical values. These examples do not represent the entire syntax and possible semantics of the ACCESS TO PROPERTIES clause.

- `GRANT ACCESS TO bob PROPERTIES USER fred;`
- `GRANT ACCESS TO PUBLIC PROPERTIES USER dbuser;`
- `GRANT ACCESS TO bob PROPERTIES USER dbuser HOME "/home/dbuser/bob";`
- `GRANT ACCESS TO bob PROPERTIES UID 101, GROUP (10011);`
- `GRANT ACCESS TO bob PROPERTIES 101, GROUP (ifx_user);`
- `GRANT ACCESS TO bob PROPERTIES USER fred, GROUP (ifx_user), AUTHORIZATION (DBSA);`

GRANT FRAGMENT statement

Use the GRANT FRAGMENT statement to assign privileges on table fragments in the local database if the table is fragmented by expression.

Syntax

(explicit id unique_778_Connect_42_grfr002) unique_778_Connect_42_grfr002 **GRANT FRAGMENT** <Fragment-Level Privileges>[] **ON** [*'owner'* .] *table* (*fragment*) **TO** { { **PUBLIC** | { | *'user'* } } [**WITH GRANT OPTION**] | *'role'* } [**AS** *'grantor'*]

Element	Description	Restrictions	Syntax
fragment	Name of a fragment	Must exist; cannot be delimited by quotation marks	Identifier on page 1231
grantor	User who can revoke the privileges	Same as for *user*	Owner name on page 1262
owner	User who owns *table*	Must be owner of *table*	Owner name on page 1262
role	Role to receive privileges	Must exist in **sysusers**	Owner name on page 1262
table	Fragmented table on which fragment privileges are granted	Must exist and must be fragmented by expression	Identifier on page 1231
user	User to whom privileges are to be granted	Must be a valid authorization identifier	Owner name on page 1262

Usage

This statement is an extension to the ANSI/ISO standard for SQL.

Use the GRANT FRAGMENT statement to grant to users (or roles) any of the Insert, Update, and Delete access privileges on individual fragments of a table. The GRANT FRAGMENT statement is valid only for tables that are fragmented according

to an expression-based distribution scheme. For an explanation of this type of fragmentation strategy, see Expression Distribution Scheme on page 35.

Fragment-Level Privileges

The keyword or keywords that follow the FRAGMENT keyword specify *fragment-level privileges*, which are a logical subset of table-level privileges:

Fragment-Level Privileges

```
"{ALL|{ | INSERT | DELETE | UPDATE}}"
```

These keywords correspond to the following fragment-level privileges:

Keyword

Effect on Grantee

ALL

Receives Insert, Delete, and Update privileges on the fragment

INSERT

Can insert rows into the fragment

DELETE

Can delete rows from the fragment

UPDATE

Can update rows in the fragment and in any columns.

Definition of Fragment-Level Authorization

In an ANSI-compliant database, the owner implicitly receives all table-level privileges on a newly created table, but no other users receive privileges.

A user who has table privileges on a fragmented table has the privileges implicitly on all fragments of the table. These privileges are not recorded in the **sysfragauth** system catalog table.

When a fragmented table is created in a database that is not ANSI compliant, the table owner implicitly receives all table-level privileges on the table, and other users (that is, PUBLIC) receive all fragment-level privileges by default. The privileges granted to PUBLIC are explicitly recorded in the **systabauth** system catalog table.

If you use the REVOKE statement to withdraw existing table-level privileges, however, you can then use the GRANT FRAGMENT statement to restore specified table-level privileges to users, roles, or PUBLIC on some subset of the fragments.

Whether or not the database is ANSI compliant, you can use the GRANT FRAGMENT statement to grant explicit Insert, Update, and Delete privileges on one or more fragments of a table that is fragmented by expression. The privileges that the GRANT FRAGMENT statement grants are explicitly recorded in the **sysfragauth** system catalog table.

The Insert, Update, and Delete privileges that are conferred on table fragments by the GRANT FRAGMENT statement are collectively known as *fragment-level privileges* or *fragment-level authority*.

Effect of Fragment-Level Authorization in Statement Validation

Fragment-level privilege enables users to execute INSERT, DELETE, and UPDATE data manipulation language (DML) statements on table fragments, even if the grantees lack Insert, Update, and Delete privileges on the table as a whole. Users who lack the table privileges can insert, delete, and update rows in authorized fragments because of the algorithm by which the database server validates DML statements. This algorithm consists of the following checks:

1. When a user executes an INSERT, DELETE, or UPDATE statement, the database server first checks whether the user has the table privileges necessary for the operation attempted. If the table privileges exist, the statement continues processing.
2. If the table privileges do not exist, the database server checks whether the table is fragmented by expression. If the table is not fragmented by expression, the database server returns an error to the user. This error indicates that the user does not have the privilege to execute the statement.
3. If the table is fragmented by expression, the database server checks whether the user holds the fragment privileges necessary for the attempted operation. If the user holds the required fragment privileges, the database server continues to process the statement. If the fragment privileges do not exist, the database server returns an error to the user. This error indicates that the user does not have the privilege to execute the statement.

Duration of Fragment-Level Privileges

The duration of fragment-level privileges is tied to the duration of the fragmentation strategy for the table as a whole.

If you drop a fragmentation strategy by means of a DROP TABLE statement or by the INIT, DROP, or DETACH clauses of an ALTER FRAGMENT statement, you also drop any privileges that exist for the affected fragments. Similarly, if you drop a fragment of a table, you also drop any privileges that exist for the fragment.

Tables that are created as a result of a DETACH or INIT clause of an ALTER FRAGMENT statement do not keep the privileges that the former fragment or fragments had when they were part of the fragmented table. Instead, such tables assume the default table privileges.

If a table on which fragment privileges are defined is changed to a table with a round-robin strategy or some other expression strategy, the fragment privileges are also dropped, and the table assumes the default table privileges.

Specifying Fragments

You can specify one fragment or a comma-separated list of fragments, with the name (or list of names) enclosed between parentheses that immediately follow the ON *table* specification. You cannot use quotation marks to delimit fragment names.

The database server issues an error if you include no fragment, or if no fragment of the specified table matches a fragment that you list.

Each *fragment* must be referenced by its name. If you did not declare an explicit identifier when you created the fragment, its name defaults to the name of the dbspace in which it resides.

After a dbspace is renamed successfully by the onspaces utility, only the new name is valid. Informix® automatically updates existing fragmentation strategies in the system catalog to substitute the new dbspace name, but you must specify the new name in GRANT FRAGMENT statement to reference a fragment whose default name is the name of a renamed dbspace.

The TO Clause

The list of one or more users or roles that follows the TO keyword identifies the grantees. You can specify the PUBLIC keyword to grant the specified fragment-level privileges to all users.

You cannot use GRANT FRAGMENT to grant fragment-level privileges to yourself, either directly or through roles.

If you enclose *user* or *role* in quotation marks, the name is case sensitive and is stored exactly as you typed it. In an ANSI-compliant database, if you do not use quotation marks around *user* or around *role*, the name is stored in uppercase letters.

The following statement grants the Insert, Update, and Delete privileges on the fragment of the **customer** table in **part1** to user **larry**:

```
GRANT FRAGMENT ALL ON customer (part1) TO larry;
```

The following statement grants the Insert, Update, and Delete privileges on the fragments of the **customer** table in **part1** and **part2** to user **millie**:

```
GRANT FRAGMENT ALL ON customer (part1, part2) TO millie;
```

To grant privileges on all fragments of a table to the same user or users, you can use the GRANT statement instead of the GRANT FRAGMENT statement. You can also use the GRANT FRAGMENT statement for this purpose.

Assume that the **customer** table is fragmented by expression into three fragments, and these fragments reside in the dbspaces named **part1**, **part2**, and **part3**. You can use either of the following statements to grant the Insert privilege on all fragments of the table to user **helen**:

```
GRANT FRAGMENT INSERT ON customer (part1, part2, part3) TO helen;

GRANT INSERT ON customer TO helen;
```

Granting Privileges to One User or a List of Users

About this task

You can grant fragment-level privileges to a single user or to a list of users.

The following statement grants the Insert, Update, and Delete privileges on the fragment of the **customer** table in **part3** to user **oswald**:

```
GRANT FRAGMENT ALL ON customer (part3) TO oswald;
```

The following statement grants the Insert, Update, and Delete privileges on the fragment of the **customer** table in **part3** to users **jerome** and **hilda**:

```
GRANT FRAGMENT ALL ON customer (part3) TO jerome, hilda;
```

Granting One Privilege or a List of Privileges

About this task

When you specify fragment-level privileges in a GRANT FRAGMENT statement, you can specify one privilege, a list of privileges, or all privileges.

The following statement grants the Update privilege on the fragment of the **customer** table in **part1** to user **ed**:

```
GRANT FRAGMENT UPDATE ON customer (part1) TO ed;
```

The following statement grants the Update and Insert privileges on the fragment of the **customer** table in **part1** to user **susan**:

```
GRANT FRAGMENT UPDATE, INSERT ON customer (part1) TO susan;
```

The following statement grants the Insert, Update, and Delete privileges on the fragment of the **customer** table in **part1** to user **harry**:

```
GRANT FRAGMENT ALL ON customer (part1) TO harry;
```

WITH GRANT OPTION Clause

As in other GRANT statements, the WITH GRANT OPTION keywords specify that the grantee can grant the same fragment-level privileges to other users. WITH GRANT OPTION is not valid if the TO clause specifies a *role* as grantee. For additional information, see WITH GRANT OPTION keywords on page 556.

The following statement grants the Update privilege on the fragment of the **customer** table in **part3** to user **george** and also gives george the right to grant the Update privilege on the same fragment to other users:

```
GRANT FRAGMENT UPDATE ON customer (part3) TO george WITH GRANT OPTION;
```

AS grantor Clause

The AS *grantor* clause of the GRANT FRAGMENT statement can specify the grantor of the privilege. You can use this clause only if you have the DBA privilege on the database. When you include the AS *grantor* clause, the database server lists the user or role who is specified as *grantor* as the grantor of the privilege in the **grantor** column of the **sysfragauth** system catalog table.

In the next example, the DBA grants the Delete privilege on the fragment of the **customer** table in the **part3** fragment to user **martha**, and uses the AS *grantor* clause to specify that user **jack** is listed in **sysfragauth** as the grantor of the privilege:

```
GRANT FRAGMENT DELETE ON customer (part3) TO martha AS jack;
```

One effect of the AS *grantor* clause in the previous example is that user **jack** can execute the REVOKE FRAGMENT statement to cancel the Delete fragment-level privilege that **martha** holds, if this GRANT FRAGMENT statement were the only source of the fragment authority of **martha** on the **customer** rows in **part3**.

Omitting the AS grantor Clause

When GRANT FRAGMENT does not include the AS *grantor* clause, the user who issues the statement is the default grantor of the specified fragment privileges.

In the next example, the user grants the Update privilege on the fragment of the **customer** table in **part3** to user **fred**. Because this statement does not specify the AS *grantor* clause, the user who issues the statement is listed by default as the grantor of the privilege in the **sysfragauth** system catalog table.

```
GRANT FRAGMENT UPDATE ON customer (part3) TO fred;
```

If you omit the AS *grantor* clause of GRANT FRAGMENT, or if you specify your own login name as the *grantor*, you can later use the REVOKE FRAGMENT statement to revoke the privilege that you granted to the specified user. For example, if you grant the Delete privilege on the fragment of the **customer** table in **part3** to user **martha** but specify user **jack** as the grantor of the privilege, user **jack** can revoke that privilege from user **martha**, but you cannot revoke that privilege from user **martha**.

The DBA, or the owner of the fragment, can use the AS clause of the REVOKE FRAGMENT statement to revoke privileges on the fragment.

INFO statement

Use the INFO statement to list the names of all the user-defined tables in the current database, or to display information about a specific table.

Syntax

This statement is an extension to the ANSI/ISO standard for SQL. You can use this statement only with .

```
INFO { TABLES | { { COLUMNS | INDEXES | STATUS } | { PRIVILEGES | ACCESS } | { FRAGMENTS | REFERENCES } } FORtable }
```

Element	Description	Restrictions	Syntax
table	Table about which you seek information	Must exist	Database Object Name on page 1225

Usage

The INFO TABLES statement lists the names of all the user-defined tables in the current database. Other keywords that can immediately follow the INFO keyword instruct to display various attributes of the *table* whose name follows the FOR keyword. To display information from more than one keyword option, issue multiple INFO statements.

The keyword options that the INFO statement supports can display the following information:

- **TABLES Keyword**

  Use TABLES (with no FOR clause) to list the identifier of every table in the current database, not including system catalog tables. Each user-defined table is listed in one of the following formats:

  - If you are the owner of the **cust_calls** table, it appears as **cust_calls**.
  - If you are not the owner of the **cust_calls** table, the authorization identifier of the owner precedes the table name, such as '**june'.cust_calls**.
- **COLUMNS Keyword**

  Use COLUMNS to display the names and data types of the columns in the specified table, showing for each column whether NULL values are allowed.
- **INDEXES Keyword**

  Use INDEXES to display the name, owner, and type of each index of the specified table, the clustered status, and listing the indexed columns.
- **FRAGMENTS Keyword**

  Use FRAGMENTS to display the fragmentation strategy and the names of the dbspaces storing the fragments of a fragmented table. If the table is fragmented with an expression-based distribution scheme, the INFO statement also shows the expressions.
- **ACCESS or PRIVILEGES Keyword**

  Use ACCESS or PRIVILEGES to display the discretionary access privileges currently held by users, roles, and the PUBLIC group for the specified table. (These two keywords are synonyms in this context.)
- **REFERENCES Keyword**

  Use REFERENCES to display the References access privilege for users who can define referential constraints on the columns of the specified table. For database-level privileges, use a SELECT statement to query the **sysusers** system catalog table.
- **STATUS Keyword**

  Use STATUS to display information about the owner, row length, number of rows and columns, creation date, and the status of audit trails for the specified table.

An alternative to using the INFO statement of SQL is to use the **Info** command of the **SQL** menu or of the **Table** menu of to display the same and additional information.

Examples

Use the following example to list the user tables in a database:

```
INFO TABLES;
```

To display information about a specific table, use the syntax:

```
INFO info_keyword FOR table
```

Here *table* is the table name and *info_keyword* is one of the seven keyword options, besides TABLES, to the INFO statement. For example, to display the names of the columns of the table **customer**, use this statement:

```
INFO COLUMNS FOR customer;
```

This example produces the following output:

```
Column name          Type             Nulls

customer_num         serial           no
fname                char(15)         yes
lname                char(15)         yes
company              char(20)         yes
address1             char(20)         yes
address2             char(20)         yes
city                 char(15)         yes
state                char(2)          yes
zipcode              char(5)          yes
phone                char(18)         yes
```

INSERT statement

Use the INSERT statement to insert one or more new rows into a table or view, or to insert one or more elements into an SQL or collection variable.

Syntax

(explicit id unique_558_Connect_42_ins002) unique_558_Connect_42_ins002(explicit id unique_558_Connect_42_ins004)

unique_558_Connect_42_ins004(explicit id unique_558_Connect_42_ins005) unique_558_Connect_42_ins005(explicit id unique_558_Connect_42_ins007)

unique_558_Connect_42_ins007(explicit id unique_558_Connect_42_ins008) unique_558_Connect_42_ins008(explicit id unique_558_Connect_42_ins009)

unique_558_Connect_42_ins009(explicit id unique_558_Connect_42_ins010) unique_558_Connect_42_ins010 **INSERT** { **INTO** { { *synonym* | *view* | *table* } [(*column*)] { <VALUES Clause>[] | <EXECUTE Routine Clause>[] | <Subset of SELECT Statement>[] } | *external* [(*column*)] [<Subset of SELECT Statement>[]] } | [**AT** *position*[][]] **INTO** <Collection-Derived Table>[] <Field Options>[] }

Field Options

" [*field*] "

" { <VALUES Clause>[] | <Subset of SELECT Statement>[] } "

Element	Description	Restrictions	Syntax
column	Column to receive new value	See Specifying Columns on page 581.	Identifier on page 1231
external	External table into which to insert data	Must exist	Database Object Name on page 1225

Element	Description	Restrictions	Syntax
field	Field of a named or unnamed ROW data type	Must already be defined in the database	CREATE ROW TYPE statement on page 286
position	Position at which to insert an element of a LIST data type	Literal integer or an INT or SMALLINT type SPL variable.	Literal Number on page 1198
synonym, *table*, *view*	Table, view, or synonym in which to insert data	Synonym or view and the table to which it points must exist	Database Object Name on page 1225

Usage

To insert data into a table, you must either own the table or have the Insert privilege for the table (see GRANT statement on page 537). To insert data into a view, you must have the required Insert privilege, and the view must meet the requirements explained in Inserting Rows Through a View on page 582.

If the table or view has data integrity constraints, the inserted rows must meet the constraint criteria. If they do not, the database server returns an error. If the checking mode is set to IMMEDIATE, all specified constraints are checked at the end of each INSERT statement. If the checking mode is set to DEFERRED, all specified constraints are *not* checked until the transaction is committed.

Specifying Columns

If you do not explicitly specify one or more columns, data is inserted into columns using the column order that was established when the table was created or last altered. The column order is listed in the **syscolumns** system catalog table.

In , you can use the DESCRIBE statement with an INSERT statement to identify the column order and the data type of the columns in a table.

The number of columns specified in the INSERT INTO clause must equal the number of values supplied in the VALUES clause or by the SELECT statement, either implicitly or explicitly. If you specify a column list, the columns receive data in the order in which you list the columns. The first value following the VALUES keyword is inserted into the first column listed, the second value is inserted into the second column listed, and so on.

If you omit a column from the column list, and the column does not have a default value associated with it, the database server places a NULL value in the column when the INSERT statement is executed.

Using the AT Clause (ESQL/C, SPL)

Use the AT clause to insert LIST elements at a specified position in a collection variable. By default, Informix® adds a new element at the end of a LIST collection.

If you specify a position greater than the number of elements in the list, the database server adds the element to the end of the list. You must specify a position value of at least `1` because the first element in the list is at position `1`.

The following SPL example inserts a value at a specific position in a list:

```
CREATE PROCEDURE test3()
   DEFINE a_list LIST(SMALLINT NOT NULL);
   SELECT list_col INTO a_list FROM table1 WHERE id = 201;
   INSERT AT 3 INTO TABLE(a_list) VALUES( 9 );
   UPDATE table1 VALUES list_col = a_list WHERE id = 201;
END PROCEDURE;
```

Suppose that before this INSERT, **a_list** contained the elements {`1,8,4,5,2`}. After this INSERT, **a_list** contains the elements {`1,8,9,4,5,2`}. The new element `9` was inserted at position `3` in the list. For more information on inserting values into collection variables, see Collection-Derived Table on page 1212.

Inserting Rows Through a View

You can insert data through a *single-table* view if you have the Insert privilege on the view. To do this, the defining SELECT statement can select from only one table, and it cannot contain any of the following components:

- DISTINCT keyword
- GROUP BY clause
- Derived value (also referred to as a virtual column)
- Aggregate value

Columns in the underlying table that are unspecified in the view receive either a default value or a NULL value if no default is specified. If one of these columns has no default value, and a NULL value is not allowed, the INSERT fails.

You can use data-integrity constraints to prevent users from inserting values into the underlying table that do not fit the view-defining SELECT statement. For further information, see WITH CHECK OPTION Keywords on page 425.

You can insert rows through a *single-table* or a *multiple-table* view if an INSTEAD OF trigger specifies valid INSERT operations in its Action clause. See #unique_516 for information on how to create INSTEAD OF triggers that insert through views.

If several users are entering sensitive information into a single table, the built-in USER function can limit their view to only the specific rows that each user inserted. The following example contains a view and an INSERT statement that achieves this effect:

```
CREATE VIEW salary_view AS
   SELECT lname, fname, current_salary FROM salary WHERE entered_by = USER;

INSERT INTO salary VALUES ('Smith', 'Pat', 75000, USER);
```

Inserting Rows with a Cursor

In , if you associate a cursor with an INSERT statement, you must use the OPEN, PUT, and CLOSE statements to carry out the INSERT operation. For databases that have transactions but are not ANSI-compliant, you must issue these statements within a transaction.

If you are using a cursor that is associated with an INSERT statement, the rows are buffered before they are written to the disk. The insert buffer is flushed under the following conditions:

- The buffer becomes full.
- A FLUSH statement executes.
- A CLOSE statement closes the cursor.
- In a database that is not ANSI-compliant, an OPEN statement implicitly closes and then reopens the cursor.
- A COMMIT WORK statement ends the transaction.

When the insert buffer is flushed, the client processor performs appropriate data conversion before it sends the rows to the database server. When the database server receives the buffer, it converts any user-defined data types and then begins to insert the rows one at a time into the database. If an error is encountered while the database server inserts the buffered rows into the database, any buffered rows that follow the last successfully inserted rows are discarded.

Inserting Rows into a Database Without Transactions

If you are inserting rows into a database with no transaction logging, you must take explicit action to restore inserted rows if the operation fails. For example, if INSERT fails after entering some rows, the successfully inserted rows remain in the table. You cannot recover automatically from a failed insert into a database for which no transaction log exists

Inserting Rows into a Database with Transactions

If you are inserting rows into a database and you are using explicit transactions, use the ROLLBACK WORK statement to undo the INSERT. If you do not execute BEGIN WORK before the INSERT, and the INSERT fails, the database server automatically rolls back any data modifications made since the beginning of the INSERT. If you are using an explicit transaction, and the INSERT fails, the database server automatically undoes the effects of the INSERT.

In an ANSI-compliant database, transactions are implicit, and all database modifications take place within a transaction. In this case, if an INSERT statement fails, use the ROLLBACK WORK statement to undo the insertions.

Tables that you create with the RAW logging type are not logged. Thus, raw tables are not recoverable, even if the database uses logging.

Rows that you insert with a transaction remain locked until the end of the transaction. The end of a transaction is either a COMMIT WORK statement, where all modifications are made to the database, or a ROLLBACK WORK statement, where none of the modifications are made to the database. If many rows are affected by a *single* INSERT statement, you can exceed the maximum number of simultaneous locks permitted. To prevent this situation, either insert fewer rows per transaction, or lock the page (or the entire table) before you execute the INSERT statement.

VALUES Clause

The VALUES clause can specify values to insert into one or more columns. When you use the VALUES clause, you can insert only one row at a time.

Each value that follows the VALUES keyword is assigned to the corresponding column listed in the INSERT INTO clause (or in column order, if a list of columns is not specified). If you are inserting a quoted string into a column, the maximum length that can be inserted without error is 256 bytes.

(explicit id unique_819_Connect_42_vacl002) unique_819_Connect_42_vacl002(explicit id unique_819_Connect_42_vacl003)
unique_819_Connect_42_vacl003(explicit id unique_819_Connect_42_vacl005) unique_819_Connect_42_vacl005(explicit id unique_819_Connect_42_vacl006)
unique_819_Connect_42_vacl006(explicit id unique_819_Connect_42_vacl007) unique_819_Connect_42_vacl007(explicit id unique_819_Connect_42_vacl008)
unique_819_Connect_42_vacl008(explicit id unique_819_Connect_42_vacl009) unique_819_Connect_42_vacl009(explicit id unique_819_Connect_42_vacl010)
unique_819_Connect_42_vacl010(explicit id unique_819_Connect_42_vacl011) unique_819_Connect_42_vacl011

VALUES Clause

" **VALUES** "

" ({ | *input_var* [{ [] :*indicator_var* | [] $ *indicator_var* }] | { **NULL** | **USER** | <Quoted String> [] | <Literal Number> [] | [] <Constant Expression> [] | <Column Expression> [] } | { <Literal Collection> [] | <Literal Row> [] | <Expression> [] | *'literal_Boolean'* | *literal_opaque* } }) "

Element	Description	Restrictions	Syntax
indicator_var	Variable to show if SQL statement returns NULL to *input_var*	See the *HCL® Informix® Enterprise Replication Guide*.	Language specific
input_var	Variable that holds value to insert. This can be a collection variable.	Can contain any value option of VALUES clause	Language specific
literal_opaque	Literal representation for an opaque data type	Must be recognized by the **input** support function of the opaque data type	See documentation of the opaque type.
literal_Boolean	Literal representation of a BOOLEAN value as a single character	Either `'t'` (TRUE) or `'f'` (FALSE)	Quoted String on page 1202

In , if you use an *input_var* variable to specify the value, you can insert character strings longer than 256 bytes into a table.

For the keywords and the types of literal values that are valid in the VALUES clause, refer to Constant Expressions on page 1040.

Considering Data Types

The value that the INSERT statement puts into a column does not need to be of the same data type as the column that receives it. These two data types, however, must be compatible. Two data types are *compatible* if the database server has some way to cast one data type to another. A *cast* is the mechanism by which the database server converts one data type to another.

The database server makes every effort to perform data conversion. If the data cannot be converted, the INSERT operation fails. Data conversion also fails if the target data type cannot hold the value that is specified. For example, you cannot insert the integer `123456` into a column defined as a SMALLINT data type because this data type cannot hold a number that large.

For a summary of the casting that the database server provides, see the *HCL® Informix® Guide to SQL: Reference*. For information on how to create a user-defined cast, see the CREATE CAST statement in this document and *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

In a database that uses a nondefault locale, if the **GL_DATETIME** environment variable has a nondefault setting, the **USE_DTENV** environment variable must be set to 1 before the INSERT statement can correctly insert localized DATETIME values into a database table, or into a view, or into an EXTERNAL table object.

Inserting Values into Serial Columns

You can insert successive numbers, explicit values, or explicit values that reset the value in a SERIAL, BIGSERIAL, or SERIAL8 column:

- To insert a consecutive serial value

  Specify a zero (0) for the serial column in the INSERT statement. In this case, the database server assigns the next highest value.
- To insert an explicit value

  Specify the nonzero value after first verifying that it does not duplicate one already in the table. If the serial column is uniquely indexed or has a unique constraint, and your value duplicates one already in the table, an error results. If the value is greater than the current maximum value, you will create a gap in the series.
- To create a gap in the series (that is, to reset the serial value)

  Specify a positive value that is greater than the current maximum value in the column.

  Alternatively, you can use the MODIFY clause of the ALTER TABLE statement to reset the next value of a serial column.

For more information, see Altering the Next Serial Value on page 141.

NULL values are not valid in serial columns.

In Informix®, inserting a serial value into a table that is part of a table hierarchy updates all tables in the hierarchy that contain the serial counter with the value that you insert. You can express this value either as zero (0) for the next highest value, or as a specific positive integer.

Inserting Values into Opaque-Type Columns

Informix® supports INSERT operations that specify literal values of opaque data types as quoted strings in the VALUES clause. You can use this syntax to insert values of opaque UDTs into columns of tables in the local database, or into columns of tables in other databases of the local instance.

Some opaque data types require special processing when they are inserted. For example, if an opaque data type contains spatial or multirepresentational data, it might provide a choice of how to store the data: inside the internal structure or, for large objects, in a smart large object.

This is accomplished by calling a user-defined support function called **assign( )**. When you execute INSERT on a table whose rows contains one of these opaque types, the database server automatically invokes the **assign( )** function for the type. The **assign( )** function can make the decision of how to store the data. For more information about the **assign( )** support function, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

Inserting Values into Collection Columns

You can use the VALUES clause to insert values into a collection column. For more information, see Collection Constructors on page 1051.

For example, suppose you define the **tab1** table as follows:

```
CREATE TABLE tab1
   (
   int1 INTEGER,
   list1 LIST(ROW(a INTEGER, b CHAR(5)) NOT NULL),
   dec1 DECIMAL(5,2)
   );
```

The following INSERT statement inserts a row into **tab1**:

```
INSERT INTO tab1 VALUES
   (
   10,
   LIST{ROW(1,'abcde'),
      ROW(POW(3,3), '=27'),
      ROW(ROUND(ROOT(126)), '=11')},
   100
   );
```

The collection column, **list1**, in this example, has three elements. Each element is an unnamed row type with an INTEGER field and a CHAR(5) field. The first element is composed of two literal values, an integer (1) and a quoted string (`abcde`). The second and third elements also use a quoted string to indicate the second field, but specify the value for the first field with an expression.

Regardless of what method you use to insert values into a collection column, you cannot insert NULL elements into the column. Thus expressions that you use cannot evaluate to NULL. If the collection that you are attempting to insert contains a NULL element, the database server returns an error.

You can also use a collection variable to insert the values of one or more collection elements into a collection column. For more information, see Collection-Derived Table on page 1212.

Inserting Values into ROW-Type Columns

Use the VALUES clause to insert values into a named or unnamed ROW type column, as in the following example:

```
CREATE ROW TYPE address_t
   (
   street CHAR(20),
   city CHAR(15),
   state CHAR(2),
   zipcode CHAR(9)
```

```
   );
CREATE TABLE employee
   (
   name ROW ( fname CHAR(20), lname CHAR(20)),
   address address_t
   );
```

The next example inserts literal values in the **name** and **address** columns:

```
INSERT INTO employee VALUES
   (
      ROW('John', 'Williams'),
      ROW('103 Baker St', 'Tracy','CA', 94060)::address_t
   );
```

INSERT uses ROW constructors to generate values for the **name** column (an unnamed ROW data type) and the **address** column (a named ROW data type). When you specify a value for a named ROW data type, you must use the CAST AS keywords or the double colon (**::**) operator, with the name of the ROW data type, to cast the value to the named ROW data type.

For the syntax of ROW constructors, see Constructor Expressions on page 1049 in the Expression segment. For information on literal values for named ROW and unnamed ROW data types, see Literal Row on page 1199.

When you use a ROW variable in the VALUES clause, the ROW variable must contain values for each field value. For more information, see Inserting into a Row Variable (ESQL/C, SPL) on page 592.

You can use host variables to insert *nonliteral* values in two ways:

- An entire ROW type into a column. Use a **row** variable in the VALUES clause to insert values for all fields in a ROW column at one time.
- Individual fields of a ROW type. To insert nonliteral values in a ROW-type column, insert the elements into a **row** variable and then specify the **collection** variable in the SET clause of an UPDATE statement.

Data Types in Distributed INSERT Operations

Distributed INSERT operations across tables in databases of different server instances can return only a subset of the data types that distributed INSERT operations can return from tables that are all in databases of the same Informix® instance.

An INSERT statement (or any other SQL data-manipulation language statement) that accesses a database of another Informix® instance can reference only the following data types:

- Built-in data types that are not opaque or complex
- BOOLEAN
- BSON
- JSON
- LVARCHAR
- DISTINCT of built-in types that are not opaque
- DISTINCT of BOOLEAN
- DISTINCT of BSON

- DISTINCT of JSON
- DISTINCT of LVARCHAR
- DISTINCT of the DISTINCT types in this list.

Cross-server distributed INSERT operations can support these DISTINCT types only if the DISTINCT types are cast explicitly to built-in types, and all of the DISTINCT types, their data type hierarchies, and their casts are defined exactly the same way in each participating database.

Cross-server DML operations cannot reference a column or expression of a complex, large-object, nor user-defined data type (UDT), nor of an unsupported DISTINCT or built-in opaque type. For additional information about the data types that Informix® supports in cross-server DML operations, see Data Types in Cross-Server Transactions on page 702.

Distributed operations that access other databases of the local Informix® instance, however, can access the cross-server data types that are listed above, and also the following data types:

- Most of the *built-in opaque data types*, as listed in Data Types in Cross-Database Transactions on page 700
- DISTINCT of the built-in types that are referenced in the line above
- DISTINCT of any of the data types that are listed in either of the two lines above
- Opaque user-defined data types (UDTs) that can be cast explicitly to built-in data types.

Cross-database INSERT operations can support these DISTINCT and opaque UDTs only if all the opaque UDTs and DISTINCT types are cast explicitly to built-in types, and all of the opaque UDTs, DISTINCT types, data type hierarchies, and casts are defined exactly the same way in each participating database.

Distributed INSERT transactions cannot access the database of another Informix® instance unless both servers define TCP/IP or IPCSTR connections in their DBSERVERNAME or DBSERVERALIASES configuration parameters and in the **sqlhosts** file or SQLHOSTS registry subkey. The requirement, that both participating servers support the same type of connection (either TCP/IP or else IPCSTR), applies to any communication between Informix® instances, even if both reside on the same computer.

Using Expressions in the VALUES Clause

With HCL Informix®, you can insert any type of expression except a column expression into a column. For example, you can insert built-in functions that return the current date, date and time, login name of the current user, or database server name where the current database resides.

The TODAY keyword returns the system date. The CURRENT or SYSDATE keyword returns the system date and time. The USER or CURRENT_USER keyword returns a string that contains the login account name of the current user. The SITENAME or DBSERVERNAME keyword returns the database server name where the current database resides. The following example uses built-in functions to insert data:

```
INSERT INTO cust_calls (customer_num, call_dtime, user_id,
                call_code, call_descr)
   VALUES (212, CURRENT, USER, 'L', '2 days');
```

For more information, see Expression on page 1003.

Inserting NULL Values

When you execute the INSERT statement, the database server inserts a NULL value into any column for which you provide no value, as well as for all columns that have no default values and that are not listed explicitly. You also can specify the NULL keyword in the VALUES clause to indicate that a column should be assigned a NULL value.

The following example inserts values into three columns of the **orders** table:

```
INSERT INTO orders (orders_num, order_date, customer_num) VALUES (0, NULL, 123);
```

In this example, a NULL value is explicitly entered in the **order_date** column, and all other columns of the **orders** table that are *not* explicitly listed in the INSERT INTO clause are also filled with NULL values.

Inserting Values into Protected Tables

In a database that uses label-based access control (LBAC), the INTO clause of the INSERT statement can reference a table that is protected by a security policy if the user holds sufficient credentials for the security policy of the label that protects the table, as well as holding the Insert privilege on the table.

A user who holds no security label, however, cannot insert data into a table that has LBAC row protection, even if the user has been granted the required exemptions from rules of the security policy, unless the row label of the protected table is specified in the VALUES clause of the INSERT statement. Data manipulation language statements can provide the row label of a protected table by calling any of three built-in functions whose first argument specifies the name of the security policy, and whose additional arguments are one of the following:

- name of the security label
- name of the IDSSECURITYLABEL column in the table.
- names of the security policy components in the label and the values of their elements

For example, the following INSERT statement calls the built-in **SECLABEL_BY_NAME** function in order to insert a new row into a table called **tab002** that is protected by a row label called **Decca** of the **MegaCorp** security policy:

```
INSERT INTO tab002
   VALUES (SECLABEL_BY_NAME('Megacorp', 'Decca'), 45, 'A.C.Debussy');
```

Whether this INSERT operation succeeds depends on whether the security credentials of the user are sufficient, relative to the component values of the **Decca** label, to enable write access to the **tab002** table.

For additional examples of INSERT statements that access protected tables by calling **SECLABEL_BY_NAME** or similar built-in functions, see Security Label Support Functions on page 1092. For general information about LBAC security policies, security labels, read and write access rules, and exemptions from those rules, see your HCL Informix® Security Guide.

Truncated CHAR Values

In a database that is not ANSI-compliant, if you assign a value to a CHAR(*n*) column or variable and the length of that value exceeds *n* characters, the database server truncates the last characters without raising an error. For example, suppose that you define this table:

```
CREATE TABLE tab1 (col_one CHAR(2);
```

The database server truncates the data values in the following INSERT statements to `"jo"` and `"sa"` respectively, but does not return a warning:

```
INSERT INTO tab1 VALUES ("john");
INSERT INTO tab1 VALUES ("sally");
```

Thus, in a database that is not ANSI-compliant, the semantic integrity of data for a CHAR(*n*) column or variable is not enforced when the value inserted or updated exceeds the declared length *n*. (But in an ANSI-compliant database, the database server issues error -1279 when truncation of character data occurs.)

Subset of SELECT Statement

As indicated in the INSERT statement on page 580 syntax diagram, not all clauses and options of the SELECT statement are available for you to use in a query within an INSERT statement. The following SELECT clauses and options are not supported in an INSERT statement:

- FIRST and LIMIT
- INTO TEMP, INTO RAW, and INTO STANDARD result table options
- UNION, UNION ALL, INTERSECT, MINUS, and EXCEPT set operators.

In an ANSI-compliant database, if this statement has a WHERE clause that does not return rows, **sqlca** returns SQLNOTFOUND (100).

If an INSERT statement that is part of a multistatement prepared object inserts no rows, **sqlca** returns SQLNOTFOUND (100) for both ANSI-compliant databases and databases that are not ANSI-compliant. In databases that are not ANSI-compliant, **sqlca** returns zero (0) if no rows satisfy the WHERE clause.

In Informix®, if you are inserting values into a supertable in a table hierarchy, the subquery can reference a subtable. If you are inserting values into a subtable in a table hierarchy, the subquery can reference the supertable if it references only the supertable. That is, the subquery must use the SELECT…FROM ONLY (*supertable*) syntax.

Execute Routine Clause

You can specify the EXECUTE FUNCTION statement (or the EXECUTE PROCEDURE statement) to insert values that a user-defined function returns

.

(explicit id unique_820_Connect_42_exrcl003) unique_820_Connect_42_exrcl003

Execute Routine Clause

" **EXECUTE** "

" { **PROCEDURE***procedure* | **FUNCTION***function* } "

" ([<Argument>[]]) "

Element	Description	Restrictions	Syntax
function, *procedure*	User-defined function or procedure to insert the data	Must exist	Database Object Name on page 1225

When you use a user-defined function to insert column values, the return values of the function must have a one-to-one correspondence with the listed columns. That is, each return value

- must be of a data type the same or compatible with the corresponding *column*,
- and its order among return values must match the *column*'s order in the column list.

For backward compatibility, Informix® can use the EXECUTE PROCEDURE keywords to execute an SPL function that was created with the CREATE PROCEDURE statement.

If the called SPL routine scans or updates the target table of the INSERT statement, the database returns an error. That is, the SPL routine cannot select data from the table into which you are inserting rows.

If a called SPL routine contains certain SQL statements, the database server returns an error. For information on which SQL statements cannot be used in an SPL routine that is called within a data manipulation statement, see Restrictions on SPL Routines in Data-Manipulation Statements on page 1293.

Number of Values Returned by SPL, C, and Java™ Functions

An SPL function can return one or more values. Make sure that the number of returned values matches the number of columns in the table or the number of columns in the column list of the INSERT statement. These columns must have data types that are compatible with the values that the SPL function returns.

An external function written in the C or Java™ language can only return *one* value. Make sure that you specify only one column in the column list of the INSERT statement. This column must have a compatible data type with the value that the external function returns. The external function can be an iterator function.

The following example shows how to insert data into a temporary table called **result_tmp** in order to output to a file the results of a user-defined function (**f_one**) that returns multiple rows:

```
CREATE TEMP TABLE result_tmp( ... );
INSERT INTO result_tmp EXECUTE FUNCTION f_one();
UNLOAD TO 'file' SELECT * FROM foo_tmp;
```

Inserting into a Row Variable (ESQL/C, SPL)

The INSERT statement does not support a row variable in the Collection- Derived-Table segment. You can use the UPDATE statement, however, to insert new field values into a row variable. For example, the following code fragment inserts a new row into the **rectangles** table (which Inserting Values into ROW-Type Columns on page 586 defines):

```
EXEC SQL BEGIN DECLARE SECTION;
   row (x int, y int, length float, width float) myrect;
EXEC SQL END DECLARE SECTION;

...
EXEC SQL update table(:myrect)
   set x=7, y=3, length=6, width=2;
EXEC SQL insert into rectangles values (12, :myrect);
```

For more information, see Updating a Row Variable (ESQL/C) on page 882.

Using INSERT as a Dynamic Management Statement

In , you can use the INSERT statement to handle situations where you need to write code that can insert data whose structure is unknown at the time you compile. For more information, refer to the dynamic management section of the *HCL® Informix® Enterprise Replication Guide*.

LOAD statement

Use the LOAD statement to insert data from an operating-system file into an existing table or view.

Syntax

Only DB-Access supports the LOAD statement.

LOAD FROM *'filename'* [**DELIMITER** *'delimiter'*] **INSERT INTO** { *table* | *view* | *synonym* } [(*column*)]

Element	Description	Restrictions	Syntax
column	Column to receive data values from *filename*	See INSERT INTO Clause on page 598.	Identifier on page 1231
delimiter	Character to separate data values in each line of the load file. Default *delimiter* is the pipe (|) symbol.	See DELIMITER Clause on page 597.	Quoted String on page 1202
filename	Path and filename of file to read. Default pathname is current directory	See LOAD FROM File on page 593.	Specific to operating system rules

Element	Description	Restrictions	Syntax
synonym, *table*, *view*	Synonym for the table in which to insert data from *filename*	*Synonym* and *table* or *view* to which it points must exist	Database Object Name on page 1225

Usage

This statement is an extension to the ANSI/ISO standard for SQL. You can use this statement only with .

The LOAD statement appends new rows to the table. It does not overwrite existing data. You cannot add a row that has the same key as an existing row.

To use the LOAD statement, you must have Insert privileges for the table where you want to insert data. For information on database-level and table-level privileges, see the GRANT statement on page 537.

In a database that uses a nondefault locale, if the **GL_DATETIME** environment variable has a nondefault setting, the **USE_DTENV** environment variable must be set to 1 before the LOAD statement can correctly insert localized DATETIME values into a database table, or into a view, or into an object that the CREATE EXTERNAL TABLE statement defined. For more information on the **GL_DATETIME**, **GL_DATE**, **DBTIME**, and **USE_DTENV** environment variables, refer to the *HCL® Informix® GLS User's Guide*.

LOAD FROM File

The LOAD FROM file contains the data to be loaded into the specified table or view. The default pathname for the load file is the current directory.

You can use the file that the UNLOAD statement creates as the LOAD FROM file. (See UNLOAD TO File on page 863 for a description of how values of various data types are represented within the UNLOAD TO file.)

If you do not include a list of columns in the INSERT INTO clause, the fields in the file must match the columns that are specified for the table in number, order, and data type.

Each line of the file must have the same number of fields. You must define field lengths that are less than or equal to the length that is specified for the corresponding column. Specify only values that can convert to the data type of the corresponding column. The following table indicates how the database server expects you to represent the data types in the LOAD FROM file (when you use the default locale, U.S. English).

Type of Data	Input Format
blank	One or more blank characters between delimiters You can include leading blanks in fields that do not correspond to character columns.
BOOLEAN	A `t` or `T` indicates a TRUE value, and an `f` or `F` indicates a FALSE value.
COLLECTIONS	Collection must have its values surrounded by braces ({ }) and a field delimiter separating each element. For more information, see Loading Complex Data Types on page 596.

Type of Data	Input Format
DATE	Character string in the following format: *mm/dd/year* You must state the month as a two-digit number. You can use a two-digit number for the year if the year is in the 20th century. (You can specify another century algorithm with the **DBCENTURY** environment variable.) The value must be an actual date; for example, February 30 is illegal. You can use a different date format if you indicate this format with the **GL_DATE** or **DBDATE** environment variable. For more information about environment variables, see the *HCL® Informix® Guide to SQL: Reference* and the *HCL® Informix® GLS User's Guide*.
DECIMAL, MONEY, FLOAT	Value that can include a leading and/or trailing currency symbol and thousands and decimal separators Your locale files or the **DBMONEY** environment variable can specify a currency format.
NULL	Nothing between the delimiters
ROW types (named or unnamed)	ROW type must have its values surrounded by parentheses and a field delimiter that separates each element. For more information, see Loading Complex Data Types on page 596.
Simple large objects (TEXT, BYTE)	TEXT and BYTE columns are loaded directly from the LOAD TO file. For more information, see Loading Simple Large Objects on page 595.
Smart large objects (CLOB, BLOB)	CLOB and BLOB columns are loaded from a separate operating-system file. The field for the CLOB or BLOB column in the LOAD FROM file contains the name of this separate file. For more information, see Loading Smart Large Objects on page 596.
Time	Character string in *year-month-day hour:minute:second.fraction* format You cannot use data type keywords or qualifiers for DATETIME or INTERVAL values. The year must be a 4-digit number, and the month must be a 2-digit number. The **DBTIME** or **GL_DATETIME** environment variable can specify other end-user formats.
User-defined data formats (opaque types)	Associated opaque type must have an import support function defined if special processing is required to copy the data in the LOAD FROM file to the internal format of the opaque type. An import binary support function might also be required for data in binary format. The LOAD FROM file data must be in the format that the import or import binary support function expects. The associated opaque type must have an assign support function if special processing is required before writing the data in the database. See Loading Opaque-Type Columns on page 597.

For more information on **DB*** environment variables, refer to the *HCL® Informix® Guide to SQL: Reference*. For more information on **GL*** environment variables, refer to the *HCL® Informix® GLS User's Guide*.

If you are using a nondefault locale, the formats of DATE, DATETIME, MONEY, and numeric column values in the LOAD FROM file must be compatible with the formats that the locale supports for these data types. For more information, see the *HCL® Informix® GLS User's Guide*.

The following example shows the contents of an input file named **new_custs**:

```
0|Jeffery|Padgett|Wheel Thrills|3450 El Camino|Suite 10|Palo Alto|CA|94306||
0|Linda|Lane|Palo Alto Bicycles|2344 University||Palo Alto|CA|94301|
    (415)323-6440
```

This data file conveys the following information:

- Indicates a serial field by specifying a zero (0)
- Uses the pipe (|), the default delimiter
- Assigns NULL values to the **phone** field for the first row and the **address2** field for the second row

  The NULL values are shown by two delimiters with nothing between them.

The following statement loads the values from the **new_custs** file into the **customer** table that **jason** owns:

```
LOAD FROM 'new_custs' INSERT INTO jason.customer;
```

If you include any of the following special characters as part of the value of a field, you must precede the character with a backslash (\ ) escape symbol:

- Backslash
- Delimiter
- Newline character anywhere in the value of a VARCHAR or NVARCHAR column
- Newline character at end of a value for a TEXT value

Do not use the backslash character (\ ) as a field separator. It serves as an escape character to inform the LOAD statement that the next character is to be interpreted as part of the data, rather than as having special significance.

Fields that correspond to character columns can contain more characters than the defined maximum allows for the field. The extra characters are ignored.

If you are loading files that contain VARCHAR data types, note the following information:

- If you give the LOAD statement data in which the character fields (including VARCHAR) are longer than the column size, the excess characters are disregarded.
- Use the backslash (\ ) to escape embedded delimiter and backslash characters in all character fields, including VARCHAR.
- Do not use the following characters as delimiting characters in the LOAD FROM file: digits (0 to 9), the letters a to f, and A to F, the backslash (\ ) character, or the NEWLINE (CTRL-J) character.

Loading Simple Large Objects

The database server loads simple large objects (BYTE and TEXT columns) directly from the LOAD FROM file. Keep the following restrictions in mind when you load BYTE and TEXT data:

- You cannot have leading and trailing blanks in BYTE fields.
- Use the backslash (****) to escape the special significance of literal delimiter and backslash characters in TEXT fields.

- Data being loaded into a BYTE column must be in ASCII-hexadecimal form. BYTE columns cannot contain preceding blanks.
- Do not use the following characters as delimiting characters in the LOAD FROM file: digits (0 to 9), the letters a to f, and A to F, the backslash (\ ) character, or the NEWLINE (CTRL-J) character.

For loading TEXT columns in a non-default locale, the database server handles any required code-set conversions for the data. See also the *HCL® Informix® GLS User's Guide*.

If you are unloading files that contain BYTE or TEXT data types, objects smaller than 10 kilobytes are stored temporarily in memory. You can adjust the 10-kilobyte setting to a larger setting with the **DBBLOBBUF** environment variable. Simple large objects that are larger than the default or the setting of **DBBLOBBUF** are stored in a temporary file. For more information about the **DBBLOBBUF** environment variable, see the *HCL® Informix® Guide to SQL: Reference*.

Loading Smart Large Objects

The database server loads smart large objects (BLOB and CLOB columns) from a separate operating-system file on the client computer. For information on the structure of this file, see Unloading Smart Large Objects on page 865.

In a LOAD FROM file, a CLOB or BLOB column value appears as follows:

```
start_off,length,client_path
```

In this format, *start_off* is the starting offset (in hexadecimal) of the smart-large-object value within the client file, *length* is the length (in hexadecimal) of the BLOB or CLOB value, and *client_path* is the pathname for the client file. No blank spaces can appear between these values.

For example, to load a CLOB value that is 512 bytes long and is at offset 256 in the **/usr/apps/clob9ce7.318** file, the database server expects the CLOB value to appear as follows in the LOAD FROM file:

```
|100,200,/usr/apps/clob9ce7.318|
```

If the whole client file is to be loaded, a CLOB or BLOB column value appears as follows in the LOAD FROM file:

```
client_path
```

For example, to load a CLOB value that occupies the entire file **/usr/apps/clob9ce7.318**, the database server expects the CLOB value to appear as follows in the LOAD FROM file:

```
|/usr/apps/clob9ce7.318|
```

In DB-Access, the USING clause is valid within files executed from DB-Access. In interactive mode, DB-Access prompts you for a password, so the USING keyword and *validation_var* are not used.

For CLOB columns, the database server handles any required code-set conversions for the data. See also the *HCL® Informix® GLS User's Guide*.

Loading Complex Data Types

In a LOAD FROM file, complex data types appear as follows:

- Collections are introduced with the appropriate constructor (SET, MULTISET, or LIST), and their elements are enclosed in braces ({ }) and separated with a comma, as follows:

  ```
  constructor{val1 , val2 , ... }
  ```

  For example, to load the SET values {1, 3, 4} into a column whose data type is SET(INTEGER NOT NULL), the corresponding field of the LOAD FROM file appears as:

  ```
  |SET{1 , 3 , 4}|
  ```

- Row types (named and unnamed) are introduced with the ROW constructor and their fields are enclosed with parentheses and separated with a comma, as follows:

  ```
  ROW(val1 , val2 , ... )
  ```

  For example, to load the ROW values (1, 'abc'), the corresponding field of the LOAD FROM file appears as:

  ```
  |ROW(1 , abc)|
  ```

Loading Opaque-Type Columns

Some opaque data types require special processing when they are inserted. For example, if an opaque data type contains spatial or multirepresentational data, it might provide a choice of how to store the data: inside the internal structure or, for large objects, in a smart large object.

This processing is accomplished by calling a user-defined support function called **assign( )**. When you execute the LOAD statement on a table whose rows contain one of these opaque types, the database server automatically invokes the **assign( )** function for the type. The **assign( )** function can make the decision of how to store the data. For more information about the **assign( )** support function, see the *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

DELIMITER Clause

Use the DELIMITER clause to specify the delimiter that separates the data contained in each column in a row in the input file. You can specify TAB (CTRL-I) or a blank space (= ASCII 32) as the delimiter symbol. You cannot use the following items as the delimiter symbol:

- Backslash (\ )
- NEWLINE character (CTRL-J)
- Hexadecimal numbers (0 to 9, a to f, A to F)

If you omit this clause, the database server checks the **DBDELIMITER** environment variable. For information about how to set the **DBDELIMITER** environment variable, see the *HCL® Informix® Guide to SQL: Reference*.

If the **DBDELIMITER** environment variable has not been set, the default delimiter is the pipe (|).

The following example specifies the semicolon (**;**) as the delimiting character. The example uses Windows™ file-naming conventions.

```
LOAD FROM 'C:\data\loadfile' DELIMITER ';'
   INSERT INTO orders;
```

INSERT INTO Clause

Use the INSERT INTO clause to specify the table, synonym, or view in which to load the new data.

You must specify the column names only if one of the following conditions is true:

- You are not loading data into all columns.
- The input file does not match the default order of the columns (the order specified when the table was created).

The INTO clause cannot specify a table object that the CREATE EXTERNAL TABLE statement defined.

The following example identifies the **price** and **discount** columns as the only columns in which to add data. The example uses Windows™ file naming conventions.

```
LOAD FROM 'C:\tmp\prices' DELIMITER ','
   INSERT INTO norman.worktab(price,discount)
```

LOCK TABLE statement

Use the LOCK TABLE statement to control access to a table by other processes.

Syntax

```
LOCK TABLE [ owner. ] { table | synonym } IN { SHARE | EXCLUSIVE } MODE
```

Element	Description	Restrictions	Syntax
owner	Owner of *synonym* or *table*	Must be the owner of the specified object	Owner name on page 1262
synonym	Synonym for the table to be locked	Synonym and the table to which it points must exist	Identifier on page 1231
table	Table to be locked	See first paragraph of Usage on page 598.	Identifier on page 1231

Usage

This statement is an extension to the ANSI/ISO standard for SQL.

You can use LOCK TABLE to lock a table if either of the following is true:

- You are the owner of the table.
- You have Select privilege on the table or on a column in the table, either from a direct grant or from a grant to PUBLIC or to your current role.

The LOCK TABLE statement fails if the table is already locked in EXCLUSIVE mode by another process, or if you request an EXCLUSIVE lock while another user has locked the same table in SHARE mode.

The SHARE keyword locks a table in *shared mode*. Shared mode gives other processes *read* access to the table but denies *write* access. Other processes cannot update or delete data if a table is locked in shared mode.

The EXCLUSIVE keyword locks a table in *exclusive mode*. This mode denies other processes both read and write access to the table. Exclusive-mode locking automatically occurs during the following statements:

- ALTER FRAGMENT
- ALTER INDEX
- ALTER TABLE
- CREATE INDEX
- DROP INDEX
- RENAME COLUMN
- RENAME TABLE
- START VIOLATIONS TABLE
- STOP VIOLATIONS TABLE
- TRUNCATE

The ONLINE keyword in some DDL operations

During certain ALTER FRAGMENT, DROP INDEX, and CREATE INDEX operations, including the ONLINE keyword can reduce the risk of run time errors when a concurrent session attempts to access the same table. For more information about the locking behavior of DDL statements that support the ONLINE keyword option, see these topics:

- Using the ONLINE keyword in ATTACH operations on page 31
- Using the ONLINE keyword in DETACH operations on page 38
- Using the ONLINE keyword in MODIFY operations on page 67
- The ONLINE keyword of CREATE INDEX on page 260
- The ONLINE keyword of DROP INDEX on page 477.

LOCK TABLE statement behavior on secondary servers

You can set an exclusive lock on a table from an updatable secondary server in a high-availability cluster. For exclusive mode locks requested from a secondary server, sessions can read the table but not update it. This behavior is similar to shared access mode on a secondary server; that is, when one session has an exclusive lock on a given table, no other session can obtain a shared or exclusive lock on that table.

On read-only secondary servers, the session issuing the LOCK TABLE statement does not lock the table and the database server does not return an error to the client.

Shared mode locks in a cluster behave the same as for a standalone server. After a LOCK TABLE statement runs successfully, users can read the table but cannot modify it until the lock is released.

Concurrent Access to Tables with Shared Locks

After the LOCK TABLE statement that specifies the IN SHARE MODE keywords executes successfully, other users can read the table but cannot modify its data until the lock is released. In databases that support transaction logging, the SELECT statement can implicitly place a shared lock on each tables that is listed in the FROM clause, in order to prevent other users from modifying those tables until the query is committed or rolled back.

Concurrent Access to Tables with Exclusive Locks

After the LOCK TABLE statement with the IN EXCLUSIVE MODE option executes successfully, no other user can obtain a lock on the specified table. When you attempt a DDL operation on that table, however, you might receive RSAM error -106 if the same table is being accessed by a concurrent session (for example, by opening a cursor). This error can also affect implicit locks that certain DDL statements place on tables automatically.

This is possible because table locks do not preclude table access. An exclusive lock prevents other users from obtaining a lock, but it cannot prevent them from opening the table for write operations that wait for the exclusive lock to be released, or for Dirty Read operations on the table. You can set the **IFX_DIRTY_WAIT** environment variable to specify that the DDL operation wait for a specified number of seconds for Dirty Read operations to commit or rollback.

When one or more rows in a table are locked by an exclusive lock, the effect on other users partly depends on their transaction isolation level. Other users in all isolation levels except the Dirty Read isolation level might encounter locking errors, such as transactions that fail because the lock was not released within a specified time limit, or a deadlock situation.

On tables where row-level locking affects some of the rows, the risk of locking conflicts can be reduced by enabling transactions to read the most recently committed version of the data in the row-level locked table, rather than waiting for the transaction that holds the lock on that row to be committed or rolled back. This can be accomplished in several different ways, including these:

- From an individual session, issue this SQL statement

  ```
  SET ISOLATION TO COMMITTED READ LAST COMMITTED;
  ```

- For all sessions using Committed Read or Read Committed isolation levels, set the USELASTCOMMITTED configuration parameter to `'ALL'` or to `'COMMITTED READ'`, or else issue the SET ENVIRONMENT USELASTCOMMITTED statement with `'ALL'` or `'COMMITTED READ'` as the session environment option.
- For all sessions using Dirty Read or Read Uncommitted isolation levels, set the USELASTCOMMITTED configuration parameter to `'ALL'` or to `'DIRTY READ'`, or else issue the SET ENVIRONMENT USELASTCOMMITTED statement with `'ALL'` or `'DIRTY READ'` as the session environment option.
- For users for whom a *user*.**sysdbopen( )** procedure is defined in the database, the DBA can define that procedure to include the SET ENVIRONMENT USELASTCOMMITTED statement with `'ALL'` or `'COMMITTED READ'` as the session environment option, and also issue the SET ISOLATION statement to set Committed Read as the isolation level.
- For users for whom no *user*.**sysdbopen( )** procedure exists in the database, the DBA can define a PUBLIC.**sysdbopen** procedure that specifies the same SET ENVIRONMENT USELASTCOMMITTED and SET ISOLATION statements.

This LAST COMMITTED isolation feature is useful only when row-level locking is in effect, rather than when another session holds an exclusive lock on the entire table. This feature is disabled for the specified table when LOCK TABLE applies a table-

level lock. See #unique_858 for more information about this LAST COMMITTED feature for concurrent access to tables in which some rows are locked by exclusive locks, and for restrictions on the kinds of tables that can support this feature.

Databases with transaction logging

If your database was created with transaction logging, the LOCK TABLE statement succeeds only if it executes within a transaction. You must issue a BEGIN WORK statement before you can execute a LOCK TABLE statement.

Transactions are implicit in an ANSI-compliant database. The LOCK TABLE statement succeeds if the specified table is not already locked by another process.

The following guidelines apply to the use of the LOCK TABLE statement within transactions:

- You cannot lock system catalog tables.
- You cannot switch between shared and exclusive table locking within a transaction. For example, once you lock the table in shared mode, you cannot upgrade the lock mode to exclusive.
- If you issue a LOCK TABLE statement before you access a row in the table, and PDQ is not in effect, no row locks are set for the table. In this way, you can override row-level locking and avoid exceeding the maximum number of locks that are defined in the database server configuration. (But if PDQ is in effect, you might run out of locks with error -134 unless the LOCKS parameter of your ONCONFIG file specifies a large enough number of locks.)
- All row and table locks release automatically after a transaction is completed. The UNLOCK TABLE statement fails in a database that uses transaction logging.
- The same user can explicitly use LOCK TABLE to lock up to 32 tables concurrently. (Use SET ISOLATION to specify an appropriate isolation level, such as Repeatable Read, if you need to lock rows from more than 32 tables during a single transaction.)

The following example shows how to change the locking mode of a table in a database that was created with transaction logging:

```
BEGIN WORK;
LOCK TABLE orders IN EXCLUSIVE MODE;
 ...
COMMIT WORK;
BEGIN WORK;
LOCK TABLE orders IN SHARE MODE;
 ...
COMMIT WORK;
```

Warning: It is recommended that you not use nonlogging tables in a transaction. If you need to use a nonlogging table in a transaction, either lock the table in exclusive mode or set the isolation level to Repeatable Read to prevent concurrency problems.

Databases without transaction logging

In a database that was created without transaction logging (by omitting the WITH LOG keywords in the CREATE DATABASE statement), table locks that were set by the LOCK TABLE statement are released after any of the following events:

- An UNLOCK TABLE statement executes.
- The user closes the database.
- The user exits from the application.

To change the lock mode on a table, release the lock with the UNLOCK TABLE statement and then issue a new LOCK TABLE statement.

The following example shows how to change the lock mode of a table in an unlogged database:

```
LOCK TABLE orders IN EXCLUSIVE MODE;
   . . .
UNLOCK TABLE orders;
   . . .
LOCK TABLE orders IN SHARE MODE;
```

Locking Granularity

The default granularity for locking a table is at the *page* level, or whatever you specify (either PAGE or ROW) in the **IFX_TABLE_LOCKMODE** environment variable, or if that is not set, by setting DEF_TABLE_LOCKMODE in the ONCONFIG file. The LOCK MODE clause of the CREATE TABLE or ALTER TABLE statement can override the default locking granularity by specifying PAGE or ROW. Only row-level locks support the LAST COMMITTED feature of Informix®.

The LOCK TABLE statement, however, always locks the entire table, overriding any other locking granularity specification for the table.

In all of these contexts, the term "lock mode" means the *locking granularity*. In the context of the SET LOCK MODE statement, however, "lock mode" refers to the behavior of the database server when a process attempts to access a row or a table that another process has locked.

MERGE statement

Use the MERGE statement to transfer data from a source table into a target table by combining UPDATE or DELETE operations with INSERT operations in a single SQL statement. You can also use this statement to join the source and target tables, and then perform only UPDATE operations, only DELETE operations, or only INSERT operations on the target table.

The MERGE statement supports the ANSI/ISO standard for SQL with Informix® extensions.

Syntax

(explicit id unique_862_Connect_42_merg201a) unique_862_Connect_42_merg201a(explicit id unique_862_Connect_42_merg202a) unique_862_Connect_42_merg202a(explicit id unique_862_Connect_42_merg203a) unique_862_Connect_42_merg203a **MERGE** [*directive*] **INTO** { *target_table* | *target_view* | *target_synonym* } [**[AS]** *alias*] **USING** { *source_table* | *source_view* | (*source_subquery*) } [**[AS]** *alias*] [(*derived_column*)] **ON** *condition* { [] <Insert Clause> | [] { <Update Clause> | <Delete Clause> } }

Update Clause

" **WHEN MATCHED THEN UPDATE** <SET Clause>[] "

Delete Clause

" **WHEN MATCHED THEN DELETE** "

Insert Clause

" **WHEN NOT MATCHED THEN INSERT** [(*column*)] <VALUES Clause>[] "

Element	Description	Restrictions	Syntax
alias	A temporary name that you declare here for the *target* or *source* table object	The source and target aliases must be different. If potentially ambiguous, the AS keyword must precede *alias*.	Identifier on page 1231
column	A column in the target object into which to insert source data	This must exist in the target object	Identifier on page 1231
condition	A Boolean condition to apply to rows in the join of the *source* and *target* tables	This can reference data values in the *source* and *target* objects	Condition on page 967
derived_column	A name that you declare here if the source object is a derived table	The SET and VALUES clauses can reference this name.	Identifier on page 1231
directive	A query optimizer directive	The directive must be valid.	Optimizer Directives on page 1246
source_table, *source_view*, *source_subquery*	A table (or the result of a query) containing data to be relocated	Object must exist. See also Restrictions on Source and Target Tables on page 608.	Database Object Name on page 1225; SELECT statement on page 690

Element	Description	Restrictions	Syntax
target_table, *target_view*, *target_synonym*	The name or synonym of a table or updatable view in which to insert, update, or delete data	See Restrictions on Source and Target Tables on page 608.	Database Object Name on page 1225

Usage

The MERGE statement of Informix® is a data manipulation language (DML) statement that joins a source table object with a target table or view. The *condition* that you specify after the ON keyword determines which rows from the source object are used in UPDATE or DELETE operations on the target, and which rows are used in INSERT operations on the target. The MERGE statement does not modify its source object.

The condition must be followed by the WHEN MATCHED THEN keywords of the Delete or the Update clause, or by the WHEN NOT MATCHED THEN keywords of the Insert clause, or by both the Update (or Delete) and Insert clauses.

- If you specify both the Update clause and the Insert clause, the MERGE statement can perform both INSERT and UPDATE operations on the target object.
- If you specify both the Delete clause and the Insert clause, the MERGE statement can perform both INSERT and DELETE operations on the target object.
- If you specify no Insert clause, no INSERT operations are performed, but the Update clause must specify an UPDATE operation (or else the Delete clause must specify a DELETE operation) on the target object for source rows that match the condition.
- If you specify no Update clause and no Delete clause, no UPDATE or DELETE operations are performed, but the Insert clause must specify an INSERT operation on the target object for source rows that do not match the condition.

The MERGE statement fails with an error if no Delete clause, no Update clause, and no Insert clause is specified.

The MERGE statement can have the following effects on the target object:

- If the Update clause is included, the MERGE statement updates rows in the target table or view according to the specifications of the SET clause with data from rows in the source table for which the condition evaluates to true.
- If the Delete clause is included, the MERGE statement deletes from the target table or view the rows for which the condition evaluates to true.
- If the Insert clause is included, the MERGE statement inserts new rows into the target table or view according to the specifications of the VALUES clause with data from rows in the source table for which the condition evaluates to false.

A single MERGE statement, however, can have no more than two of these three effects, because the Delete clause and the Update clause are mutually exclusive.

For operations on large tables, make sure that these resources are available on your system:

- A sufficient number of locks
- Sufficient temporary dbspace storage for the intermediate join results
- Sufficient dbspace storage for the results of the MERGE statement.

In a high-availability cluster configuration, you can issue the MERGE statement from a primary server or from an updatable secondary server.

Optimizer Directives and Subqueries

You can optionally specify one or more query optimizer directives after the MERGE keyword, such as access method directives, join order directives, and join method directives to specify how the source and target tables are joined. The goal-oriented directives like EXPLAIN and AVOID_EXECUTE are also valid in the MERGE statement.

Within the MERGE statement, subqueries can also include optimizer directives to control other aspects of the execution plan. Subqueries are valid in the following contexts in the MERGE statement:

- In the *condition* of the ON clause
- In the SET clause of the Update clause
- In the VALUES clause of the Insert clause
- In the USING clause if it specifies a source query, which can include a subquery in any context where the SELECT statement supports a subquery.

The MERGE statement fails with an error, however, if it includes a subquery that references the target table.

In a database that supports external directives, the query optimizer can also apply external directives to the outer join of the source and target tables, or to subqueries within the MERGE statement.

The ON Condition

The *condition* that follows the ON keyword specifies a join filter for the source and target table objects. This ON clause filter determines the matched rows and unmatched rows in the MERGE statement, based on the outer join of the target and source tables.

- If the MERGE statement includes the Update clause, and the ON clause condition evaluates to true, then the corresponding rows are updated in the target.
- If the MERGE statement includes the Delete clause, and the ON clause condition evaluates to true, then the corresponding rows are deleted from the target.
- If the MERGE statement includes the Insert clause, and the ON clause condition evaluates to false, then the corresponding source rows are inserted into the target.

Update operations of the MERGE statement on rows that match the condition obey the UPDATE statement rules for the SET clause. For details of the syntax for specifying the updated values in the target table, see SET Clause on page 873.

Delete operations of MERGE on rows that match the condition obey the DELETE statement rules. For details of deleting values from the target table, see #unique_620.

Insert operations on rows that do not match the condition obey the INSERT statement rules for the VALUES clause. For details of the syntax for specifying the inserted values in the target table, see VALUES Clause on page 583.

Error Handling

If an error occurs while the MERGE statement is executing, the entire statement is rolled back.

For databases that support transaction logging, you can include error-handling logic that includes the ROLLBACK TO SAVEPOINT statement in a transaction that includes the MERGE statement and that defines one or more savepoints. After a partial rollback of the transaction to a savepoint, the effects of the INSERT, DELETE, or UPDATE operations of the MERGE statement persist in the target table if the MERGE statement precedes the savepoint in the lexical order of statements for that savepoint level of the transaction. The effects of MERGE are rolled back, however, if the MERGE statement follows the specified savepoint within the transaction.

In an ANSI-compliant database, data manipulation language (DML) statements are always in a transaction. These databases do not support the MERGE statement outside a transaction.

Constraint Checking

Enabled data-integrity constraints on the target object are enforced in MERGE operations.

- If the checking mode is set to DEFERRED, the constraints are not checked until after the transaction is committed.
- If the constraint-checking mode for the target table is set to IMMEDIATE, then unique and referential constraints are checked after all the UPDATE (or DELETE) and INSERT operations are complete. The NOT NULL and check constraints are checked during the UPDATE, DELETE, and INSERT operations.

For information on setting the constraint-checking mode, see the topic SET Transaction Mode statement on page 841.

If a referential constraint on the target table was defined with the ON DELETE CASCADE keywords, the DELETE clause of the MERGE statement also performs cascading deletes on rows of child tables of the target table.

A Delete merge fails, however, if an enabled referential constraint has established a parent-child relationship between the target and source tables, if the constraint was defined with the ON DELETE CASCADE keywords. The MERGE statement cannot perform cascading deletes on rows of its source table. For more information, see the topic Restrictions on DELETE When Tables Have Cascading Deletes on page 454.

If the START VIOLATIONS statement has defined an active violations table on the target table, then the MERGE statement can have the following effects on the target, violations, and diagnostic tables:

- The conforming rows in the target table that match the join condition are either deleted or updated.
- The target table also receives the conforming unmatched rows that MERGE successfully inserts.
- The violation table receives the nonconforming rows.
- A diagnostic table receives information about why the nonconforming rows failed to satisfy a constraint or a unique index during operations of the MERGE statement on the target table.

To enable a violations table and a diagnostic table on the target table, the SET Database Object Mode statement must set the constraints or unique indexes of the target table to ENABLED or FILTERING mode. For more information, see the topics Relationship to the SET Database Object Mode statement on page 845 and SET Database Object Mode statement on page 784.

Using the MERGE Statement with Triggers

The target object can be a table on which an Update, Delete, or Insert trigger is defined. If both an Update trigger and an Insert trigger (or both a Delete trigger and an Insert trigger) are enabled on the target table, MERGE can act as the triggering event for both triggers, if the MERGE statement performs both UPDATE (or DELETE) and INSERT operations on the target.

If the MERGE statement includes operations that activate both Update (or Delete) and Insert triggers, the BEFORE trigger actions of both triggers are executed when the MERGE operation starts. Similarly, the AFTER trigger actions of both triggers are executed at the end of the MERGE operation. The FOR EACH ROW trigger actions are activated for each row processed.

Just as for any DML statement, the database server treats all the triggers that are activated by the same MERGE statement as a single trigger, and the resulting trigger action is the merged-action list. All the rules that govern a trigger action apply to the merged list as one list, and no distinction is made between the two original triggers. For more information, see Actions of Multiple Triggers on page 394.

The target object, however, cannot be a view on which an enabled INSTEAD OF trigger is defined. Before you can use that view as the target of a MERGE statement, you must disable or drop the INSTEAD OF trigger.

In the definition of a trigger, the MERGE statement cannot be specified directly as a triggered action. An SPL trigger routine that is called in a triggered action, however, can issue the MERGE statement.

Security Policies and Secure Auditing

If the source object or any of its columns is protected by a label-based access control (LBAC) security policy, the user who issues the MERGE statement must have a security label (or must hold a security policy exemption) that provides sufficient credentials to read the source table in MERGE operations.

If the target object or any of its columns is protected by a label-based security policy, the user who issues the MERGE statement must have a security label (or must hold a security policy exemption) that provides sufficient credentials to write in the target object columns that the SET clause or the VALUES clause specifies, or to delete rows from the target that include protected data.

If both the source and the target table are protected, they must be protected by the same security policy. The MERGE statement cannot join tables that are protected by different LBAC security policies.

On Informix® instances that use the secure-auditing facility to record activity that could potentially alter or reveal data or the auditing configuration, no specific audit event mnemonic is defined in audit trails for the MERGE statement:

- Activities specified by the Delete clause are recorded as DELETE events.
- Activities specified by the Insert clause are recorded as INSERT events.
- Activities specified by the Update clause are recorded as UPDATE events.

Restrictions on Source and Target Tables

Which table objects can be the source or target of the MERGE statement depends on attributes of the table object, and on what access privileges are held by the user who issues the MERGE statement.

The target table must be local to the database to which the current session is connected, but you can specify a remote a table as the source table, or in subqueries of the SET clause for UPDATE operations, and in subqueries of the VALUES clause for INSERT operations.

Sections that follow identify additional restrictions on the source and target tables.

Restrictions on the Source Table

The source object can be the name or synonym of a STANDARD, RAW, TEMP, EXTERNAL, or collection-derived table, or a view. It can be in the same database as the target object, or in a different database of the local Informix® instance, or it can be a remote table that is managed by a different Informix® instance.

If the source is a collection-derived table that is defined by the result of a query, the USING clause can declare names for derived columns that the SET and VALUES clauses of the MERGE statement can reference.

The user who issues the MERGE statement must hold the Connect access privilege (or a higher privilege) on the database of the source object, and must also hold the Select privilege (or a higher privilege) on the source object. The user can be granted these access privileges individually, or can hold them as a member of the PUBLIC group, or through the current or default role of the user, if the role or PUBLIC holds those privileges.

If the source object or any of its columns is protected by a label-based security policy, the user who issues the MERGE statement must have a security label (or must hold a security policy exemption) that provides sufficient credentials to read the source object. If the credentials of the user are insufficient to read protected columns, according to the standard label-based access control (LBAC) rules, then the MERGE statement can process only a subset of the source data. If this subset is empty, the MERGE statement cannot insert any values from the source object into the target table.

The following restrictions apply to the source table object:

- The source cannot be a view on which an enabled SELECT trigger is defined.
- The source cannot be a typed table in the same table hierarchy as the target table.
- In a Delete merge, the source cannot have a child-table relationship with the target, as defined by an enabled referential constraint, if that constraint was defined with the ON DELETE CASCADE keywords. (Child-table relationships have no effect on the Delete merge, however, unless a target table constraint specifies cascading deletes.)

Restrictions on the Target Table

The target table object must be in a database of the same Informix® instance to which the current session is connected. It can be the name or synonym of a STANDARD, RAW, or TEMP table, or an updatable view. If the target is a supertable within a table hierarchy, the Delete clause also deletes the corresponding rows in all the subtables of the target table.

The user who issues the MERGE statement must hold the Connect access privilege (or a higher privilege) on the database of the target object, and must also hold the Insert privilege and the Update or Delete privilege on the target object, if the MERGE statement includes the corresponding Insert, Update, or Delete clause.

The following restrictions apply to the target table of the MERGE statement. If that table has any of the following attributes, the MERGE operation returns an error.

- The target cannot be a typed table in the same table hierarchy as the source table.
- The target cannot be a Virtual Table Interface (VTI) table.
- The target cannot be an object that the CREATE EXTERNAL TABLE statement defined.
- The target cannot be in a database of a remote Informix® instance.
- The target cannot be a system catalog table.
- The target cannot be a view on which an enabled INSTEAD OF trigger is defined.
- The target cannot be a read-only view.
- The target cannot be a pseudo-table (a memory-resident object in a system database, such as the **sysmaster** or **sysadmin** databases).
- The target cannot be a data source of any subquery of the same MERGE statement, including subqueries in the ON clause, in the SET clause, or in the VALUES clause.
- If the MERGE statement includes the DELETE clause, the target cannot have a parent-table relationship with the source table, if this relationship is defined by an enabled referential constraint that specifies the ON DELETE CASCADE keywords.

Restriction on the combined row length

The source table and the target table in the MERGE statement cannot have a total combined row length (= row size of source table + row size of target table) greater than 32,767 bytes. Otherwise, the MERGE statement fails with an error, as in the following example:

```
CREATE TABLE t1
       (f1  INT,
        f2  VARCHAR(10),
        lv1 LVARCHAR(5000),
        lv2 LVARCHAR(4000),
        lv3 LVARCHAR(8000));
CREATE TABLE t2
       (f1  INT,
        f2  VARCHAR(10),
        lv1 LVARCHAR(5000),
        lv2 LVARCHAR(4000),
        lv3 LVARCHAR(8000));

INSERT INTO t1 (f1,f2) VALUES (1,'t1 1');
INSERT INTO t1 (f1,f2) VALUES (2,'t1 2');
INSERT INTO t1 (f1,f2) VALUES (3,'t1 3');
INSERT INTO t1 (f1,f2,lv1) VALUES (7,'t1 7',
   'looooooooooooooooong');

INSERT INTO t2 (f1,f2) VALUES (3,'t2 3');
INSERT INTO t2 (f1,f2) VALUES (4,'t2 4');
INSERT INTO t2 (f1,f2) VALUES (5,'t2 5');
```

```
INSERT INTO t2 (f1,f2) VALUES (6,'t2 6');

MERGE INTO t2 AS o  USING t1 AS n ON o.f1 = n.f1
   WHEN NOT MATCHED THEN INSERT ( o.f1,o.f2)
      VALUES ( n.f1,n.f2);
```

The MERGE statement above fails, because the sum of the row lengths of the source and target tables exceeds the upper limit of 32,767 bytes.

For MERGE operations that include only the INSERT clause (but no DELETE clause nor UPDATE clause), you can circumvent this row length limit by replacing the MERGE statement with INSERT INTO . . . SELECT statements. For the same tables and data values in the MERGE example above, the following INSERT statements run successfully:

```
INSERT INTO t2(f1, f2)
   SELECT t1.f1, t1.f2 FROM t1
      WHERE NOT EXISTS
         (SELECT f1, f2 FROM t2
            WHERE t2.f1 = t1.f1);

INSERT INTO t2(f1,f2)
   SELECT t1.f1, t1.f2 FROM t1
      LEFT JOIN t2 ON t1.f1 = t2.f1
         WHERE t2.f1 IS NULL;
```

After the two INSERT INTO . . . SELECT operations, table **t2** contains what the row size restriction prevented the previous MERGE example from returning.

Restrictions on distributed MERGE statements

If the source and target tables are not in the same database, both databases must satisfy the compatibility requirements for cross-database and cross-server DML operations:

- Both databases must be of release versions that support all the data types in the source table and in the target table.
- If one database is ANSI-compliant, the other must also be ANSI-compliant.
- If one database is not ANSI-compliant but uses explicit transaction logging, the other must also support explicit transaction logging. (In this case, their buffered or unbuffered logging modes need not match.)
- If one database does not support transaction logging, the other also must not.
- Both databases must have the same NLSCASE sensitivity setting.

A distributed MERGE statement cannot, for example, specify a source table in a case-sensitive database and a target table in a database created as NLSCASE INSENSITIVE, whether or not either table includes NCHAR or NVARCHAR columns.

Distributed MERGE transactions cannot access the database of another Informix® instance unless both servers define TCP/IP or IPCSTR connections in their DBSERVERNAME or DBSERVERALIASES configuration parameters and in the **sqlhosts** file or SQLHOSTS registry subkey. The requirement, that both participating servers support the same type of connection (either TCP/IP or else IPCSTR), applies to any communication between Informix® instances, even if both reside on the same computer.

Data types valid in distributed MERGE statements

When the source table and the target tables are in the same database, MERGE statements can access all the data-type categories that the Data Type on page 985 topics describe, including BYTE and TEXT objects, OPAQUE and DISTINCT types, complex types, and user-defined types (UDTs).

A MERGE statement whose source table is in a database of another Informix® server instance, however, is subject to the restrictions on column data types in other distributed data manipulation language (DML) operations. A cross-server MERGE statement can reference columns with only the following subset of data types:

- Built-in data types that are not opaque or complex
- BOOLEAN
- BSON
- JSON
- LVARCHAR
- DISTINCT of built-in types that are not opaque
- DISTINCT of BOOLEAN
- DISTINCT of BSON
- DISTINCT of JSON
- DISTINCT of LVARCHAR
- DISTINCT of the DISTINCT types in this list.

Cross-server distributed MERGE operations can support these DISTINCT types only if the DISTINCT types are cast explicitly to built-in types, and all of the DISTINCT types, their data type hierarchies, and their casts are defined exactly the same way in the source database and in the target database.

Cross-server operations can also return the IDSSECURITYLABEL data type from a table protected by row-level LBAC security, if the user issuing the MERGE statement holds sufficient LBAC credentials. Accessing a table protected only by column-level LBAC security requires similar credentials, but in that case the table has no IDSSECURITYLABEL column.

Cross-server DML operations cannot reference a column or expression of a complex, large-object, nor user-defined data type (UDT), nor of an unsupported DISTINCT or built-in opaque type. For additional information about the data types that Informix® supports in cross-server DML operations, see Data Types in Cross-Server Transactions on page 702.

Distributed MERGE operations in which the source table and the target table are in a different databases of the same Informix® server instance can access the same data types that are listed above for cross-server MERGE statements. They can also access the following additional data types that are not supported in cross-server MERGE operations:

- Most of the *built-in opaque data types*, as listed in Data Types in Cross-Database Transactions on page 700
- DISTINCT of the built-in types that are referenced in the line above
- DISTINCT of any of the data types that are listed in either of the two lines above
- Opaque UDTs that can be cast explicitly to built-in data types.

Cross-database INSERT operations can support these DISTINCT and opaque UDTs only if all the opaque UDTs and DISTINCT types are cast explicitly to built-in types, and all of the opaque UDTs, DISTINCT types, data type hierarchies, and casts are defined exactly the same way in each participating database.

If the target and source tables are in different databases, whether of the same server instance or of different server instances, the MERGE statement fails if it references a table, view, or synonym that includes a column of any of the following opaque or complex data types:

- IMPEXP, IMPEXPBIN, LOLIST, or SENDRECV built-in opaque type
- DISTINCT of any of those opaque data types
- Any complex type, including COLLECTION, LIST, MULTISET, or SET, and named or unnamed ROW types.

Handling Duplicate Rows

While MERGE is executing, the same row in the target table cannot be updated or deleted more than once. No attempt is made to update or delete any row in the target that did not already exist before the MERGE statement was executed. That is, there are no updates or deletes of rows that the same MERGE statement inserted into the target.

The following example of the MERGE statement uses the transaction table **new_sale** as the source table from which to insert or update rows in the fact table **sale**. The join condition in this example tests whether the **new_sale.cust_id** column value matches the **sale.cust_id** column value.

```
MERGE INTO sale USING new_sale AS n
   ON sale.cust_id = n.cust_id
   WHEN MATCHED THEN UPDATE
      SET sale.salecount = sale.salecount + n.salecount
   WHEN NOT MATCHED THEN INSERT (cust_id, salecount)
      VALUES (n.cust_id, n.salecount);
```

To execute this MERGE statement, the database server joins the target and source tables, and then applies the specified equality condition to process the result of the join:

- For rows that satisfy the condition (because the **sale.cust_id** value matches the **new_sale.cust_id** value), MERGE updates the **sale.salecount** column value, according to the SET clause specification.
- For rows that do not satisfy the condition (because no row in the **sale** table has the same **cust_id** value as **new_sale.cust_id**), MERGE inserts new rows containing the **new_sale.cust_id** and **new_sale.salecount** values into the **sale** table, according to the VALUES clause specification.

For the MERGE statement in the previous example, suppose that the **sale** target table contains the two records and that the **new_sale** source table contains the three records.

Table 9. Records in 'sale' Table

cust_id	sale_count
Tom	129
Julie	230

Table 10. Records in 'new_sale' Table

cust_id	sale_count
Tom	20
Julie	3
Julie	10

When merging **new_sale** into **sale** by specifying the expression `sale.cust_id = new_sale.cust_id` as the matching condition, the MERGE statement returns an error, because it attempts to update one of the records in the **sale** target table more than once.

Data Types in Distributed MERGE Operations

If the source table or view (or any table object referenced in the source query) specifies a table object in a database of a Informix® instance other than the local instance that manages the database of the target table, the MERGE statement can access columns of only the following data types in the remote database:

- Built-in data types that are not opaque
- BOOLEAN
- LVARCHAR
- DISTINCT of the built-in data types that are not opaque
- DISTINCT of BOOLEAN
- DISTINCT of LVARCHAR
- DISTINCT of any DISTINCT data type that appears in this list.

Cross-server distributed MERGE operations can support these DISTINCT types only if the DISTINCT types are cast explicitly to built-in types, and all of the DISTINCT types, their data type hierarchies, and their casts are defined exactly the same way in each participating database. For additional information about the data types that Informix® supports in cross-server DML operations, see Data Types in Cross-Server Transactions on page 702.

MERGE operations cannot access a database of another Informix® instance unless both server instances support either a TCP/IP or an IPCSTR connection, as defined in their DBSERVERNAME or DBSERVERALIASES configuration parameters and in the **sqlhosts** file or SQLHOSTS registry subkey. This connection-type requirement applies to any communication between Informix® instances, even if both database servers reside on the same computer.

Distributed MERGE operations that access table objects in other databases of the local Informix® instance, however, can access all of the cross-server data types in the preceding list, and these additional data types:

- Most built-in opaque data types, as listed in Data Types in Cross-Database Transactions on page 700
- DISTINCT of the same built-in opaque types
- DISTINCT of any of the data types in either of the two preceding lines
- Opaque user-defined data types (UDTs) that are explicitly cast to built-in data types.

The MERGE statement also supports Distributed Relational Database Architecture™ (DRDA®) protocols in common client APIs. For the Informix® data types that MERGE can return from a remote database through DRDA® protocols, see the *HCL® Informix® Administrator's Guide* for lists of the Informix® data types that are supported (and that are not supported) by DRDA®.

Examples of MERGE Statements

Examples in this section include MERGE statements that illustrate join conditions and various DML operations on the result set of the join.

Examples

The following MERGE statement includes the Update and Insert clauses, and uses an equality predicate as the join condition:

```
MERGE INTO customer c
   USING ext_customer e
   ON c.customer_num=e.customer_num
WHEN MATCHED THEN
   UPDATE SET c.fname = e.fname,
            c.lname = e.lname,
            c.company = e.company,
            c.address1 = e.address1,
            c.address2 = e.address2,
            c.city = e.city,
            c.state = e.state,
            c.zipcode = e.zipcode,
            c.phone = e.phone
WHEN NOT MATCHED THEN
   INSERT (c.fname, c.lname, c.company, c.address1, c.address2,
            c.city, c.state, c.zipcode, c.phone)
       VALUES
          (e.fname, e.lname, e.company, e.address1, e.address2,
           e.city, e.state, e.zipcode, e.phone);
```

The next example specifies multiple predicates in the ON clause:

```
MERGE INTO customer c
   USING ext_customer e
   ON c.customer_num=e.customer_num
      AND c.fname=e.fname AND c.lname=e.lname
WHEN MATCHED THEN
   UPDATE SET c.fname = e.fname,
            c.lname = e.lname,
            c.company = e.company,
            c.address1 = e.address1,
            c.address2 = e.address2,
            c.city = e.city,
            c.state = e.state,
            c.zipcode = e.zipcode,
            c.phone = e.phone
WHEN NOT MATCHED THEN
   INSERT
        (c.fname, c.lname, c.company, c.address1, c.address2,
         c.city, c.state, c.zipcode, c.phone)
        VALUES
```

```
        (e.fname, e.lname, e.company, e.address1, e.address2,
         e.city, e.state, e.zipcode, e.phone);
```

The following MERGE statement performs an Update join, with no Insert clause:

```
MERGE INTO customer c
USING ext_customer e
ON c.customer_num=e.customer_num
WHEN MATCHED THEN
UPDATE SET c.fname = e.fname,
           c.lname = e.lname,
           c.company = e.company,
           c.address1 = e.address1,
           c.address2 = e.address2,
           c.city = e.city,
           c.state = e.state,
           c.zipcode = e.zipcode,
           c.phone = e.phone ;
```

The following MERGE statement includes only the Delete clause after the join condition:

```
MERGE INTO customer c
   USING ext_customer e
      ON c.customer_num=e.customer_num
      WHEN MATCHED THEN
         DELETE ;
```

The next MERGE example includes only the Insert clause:

```
MERGE INTO customer c
USING ext_customer e
ON c.customer_num=e.customer_num AND c.fname=e.fname
   AND c.lname=e.lname
WHEN NOT MATCHED THEN
   INSERT
      (c.fname, c.lname, c.company, c.address1, c.address2,
       c.city, c.state, c.zipcode, c.phone)
       VALUES
        (e.fname, e.lname, e.company, e.address1, e.address2,
         e.city, e.state, e.zipcode, e.phone);
```

The next example illustrates that the WHEN MATCHED and WHEN NOT MATCHED specifications can appear in any order:

```
MERGE INTO customer c
   USING ext_customer e
   ON c.customer_num=e.customer_num AND c.fname=e.fname AND c.lname=e.lname
WHEN NOT MATCHED THEN
     INSERT
        (c.fname, c.lname, c.company, c.address1, c.address2,
         c.city, c.state, c.zipcode, c.phone)
     VALUES
        (e.fname, e.lname, e.company, e.address1, e.address2,
         e.city, e.state, e.zipcode, e.phone)
WHEN MATCHED THEN UPDATE
       SET c.fname = e.fname,
           c.lname = e.lname,
           c.company = e.company,
```

```
           c.address1 = e.address1,
           c.address2 = e.address2,
           c.city = e.city,
           c.state = e.state,
           c.zipcode = e.zipcode,
           c.phone = e.phone ;
```

The following MERGE statement specifies as its source a derived table that the query in the USING clause defines:

```
MERGE INTO customer c
   USING (SELECT * from ext_customer e1, orders e2
          WHERE e1.customer_num=e2.customer_num ) e
   ON c.customer_num=e.customer_num AND c.fname=e.fname
      AND c.lname=e.lname
   WHEN NOT MATCHED THEN
      INSERT (c.fname, c.lname, c.company, c.address1, c.address2,
              c.city, c.state, c.zipcode, c.phone)
      VALUES (e.fname, e.lname, e.company, e.address1, e.address2,
              e.city, e.state, e.zipcode, e.phone)
   WHEN MATCHED THEN
      UPDATE SET c.fname = e.fname,
           c.lname = e.lname,
           c.company = e.company,
           c.address1 = e.address1,
           c.address2 = e.address2,
           c.city = e.city,
           c.state = e.state,
           c.zipcode = e.zipcode,
           c.phone = e.phone ;
```

OPEN statement

Use the OPEN statement to activate a cursor.

Syntax

(explicit id unique_868_Connect_42_open002) unique_868_Connect_42_open002(explicit id unique_868_Connect_42_open003)

unique_868_Connect_42_open003 **OPEN** { *cursor_id* | *cursor_id_var* } [**USING** { *parameter_var* | **SQL DESCRIPTOR** { *'descriptor '* | *descriptor_var* } **DESCRIPTOR***sqlda_pointer* }] [**WITH REOPTIMIZATION**]

Element	Description	Restrictions	Syntax
cursor_id	Name of a cursor	Must have been declared by the DECLARE statement	Identifier on page 1231
cursor_id_var	Host variable = *cursor_id*	Must be a character data type	Language specific
descriptor	Name of a system-descriptor area	Must have been allocated	Quoted String on page 1202
descriptor_var	Host variable that identifies the system-descriptor area	System-descriptor area must have been allocated	Quoted String on page 1202

Element	Description	Restrictions	Syntax
parameter_var	Host variable whose contents replace a question (?) mark placeholder in a prepared SQL statement	Must be a character or collection data type	Language specific
sqlda_pointer	Pointer to **sqlda** structure defining data type and memory location of values to replace question (?) marks in a prepared statement	Cannot begin with a dollar ($) sign nor with a colon (:). You must use an **sqlda** structure with dynamic SQL statements.	DESCRIBE statement on page 459

Usage

Use this statement with or with SPL.

A *cursor* is an identifier associated with an SQL statement that returns an ordered set of values. The OPEN statement activates a cursor that the DECLARE statement defined.

Cursor can be classified by their associated SQL statements:

- A *Select cursor*: a cursor that is associated with a SELECT statement
- A *Function cursor*: a cursor that is associated with the EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement
- An *Insert cursor*: a cursor that is associated with the INSERT statement
- A *Collection cursor*: a Select or Insert cursor that operates on a collection variable.

In UDRs written in the SPL language, the OPEN statement can reference only Select or Function cursors, and these must specify the identifier of the cursor, rather than a variable that stores the *cursor_id*. The OPEN statement cannot reference a direct cursor that the FOREACH statement of SPL has declared.

The specific actions that the database server takes differ, depending on the statement with which the cursor is associated. In ESQL/C, when you associate one of the previous statements with a cursor directly (that is, you do not prepare the statement and associate the statement identifier with the cursor), the OPEN statement implicitly prepares the statement. (This is not a feature of OPEN in SPL routines, where the DECLARE statement associates a cursor with the identifier of an existing prepared statement, rather than directly with SQL statement text.)

In an ANSI-compliant database, you receive an error code if you try to open a cursor that is already open.

Opening an Update Cursor Inside a Transaction

If you are working in a database with explicit transactions, you must open an update cursor within a transaction. This requirement is waived if you declared the cursor using the WITH HOLD option.

Opening a Function Cursor

When you open a Function cursor, the EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement is passed to the database server with any values that are specified in the USING clause.

The values in the USING clause are passed as arguments to the user-defined function. This user-defined function must be declared to accept values. (If the statement was previously prepared, the statement was passed to the database server when it was prepared.) The database server executes the function to the point where it returns the first set of values.

The following example illustrates a simple OPEN statement in :

```
EXEC SQL declare s_curs cursor for
   execute function new_func(arg1,arg2)
   into :ret_val1, :ret_val2;
EXEC SQL open s_curs;
```

Reopening a Select or Function Cursor

The database server evaluates the values that are specified in the USING clause of the OPEN statement only when it opens a Select cursor or Function cursor. While the cursor is open, subsequent changes to program variables in the USING clause do not change the active set of the cursor.

In a database that is ANSI-compliant, you receive an error code if you try to open a cursor that is already open.

In a database that is not ANSI-compliant, a subsequent OPEN statement closes the cursor and then reopens it. When the database server reopens the cursor, it creates a new active set, based on the current values of the variables in the USING clause. If the variables have changed since the previous OPEN statement, reopening the cursor can generate an entirely different active set.

Even if the values of the variables are unchanged, the values in the active set can be different, as in the following situations:

- If the user-defined function takes a different execution path from the previous OPEN statement on a Function cursor
- If data in the table was modified since the previous OPEN statement on a Select cursor

The database server can process most queries dynamically, without pre-fetching all rows when it opens the Select or Function cursor. Therefore, if other users are modifying the table at the same time that the cursor is being processed, the active set might reflect the results of these actions.

For some queries, the database server evaluates the entire active set when it opens the cursor. These queries include those with the following features:

- Queries that require sorting: those with an ORDER BY clause or with the DISTINCT or UNIQUE keyword
- Queries that require hashing: those with a join or with the GROUP BY clause

For these queries, any changes that other users make to the table while the cursor is being processed are not reflected in the active set.

Opening an Insert Cursor (ESQL/C)

When you open an Insert cursor, the cursor passes the INSERT statement to the database server, which checks the validity of the keywords and column names. The database server also allocates memory for an insert buffer to hold new data. (See DECLARE statement on page 433.)

An OPEN statement for a cursor that is associated with an INSERT statement cannot include a USING clause.

Example of Opening an Insert Cursor

The following example illustrates an OPEN statement with an Insert cursor:

```
EXEC SQL prepare s1 from
   'insert into manufact values ('npr', 'napier')';
EXEC SQL declare in_curs cursor for s1;
EXEC SQL open in_curs;
EXEC SQL put in_curs;
EXEC SQL close in_curs;
```

Reopening an Insert Cursor

When you reopen an Insert cursor that is already open, you effectively flush the insert buffer; any rows that are stored in the insert buffer are written into the database table. The database server first closes the cursor, which causes the flush and then reopens the cursor. For information about how to check errors and count inserted rows, see Error Checking on page 640.

In an ANSI-compliant database, you receive an error code if you try to open a cursor that is already open.

Opening a Collection Cursor (ESQL/C)

You can declare both Select and Insert cursors on collection variables. Such cursors are called *Collection cursors*. You must use the OPEN statement to activate these cursors.

Use the name of a collection variable in the USING clause of the OPEN statement. For more information on the use of OPEN ... USING with a collection variable, see Fetching from a Collection Cursor on page 520 and Inserting into a Collection Cursor on page 639.

USING Clause

The USING clause is required when the cursor is associated with a prepared statement that includes question-mark (?) placeholders, as follows:

- A SELECT statement with input parameters in its WHERE clause
- An EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement with input parameters as arguments to its user-defined function
- An INSERT statement with input parameters in its VALUES clause (in ESQL/C).

In SPL routines, you must specify these parameters as SPL variables.

In ESQL/C, you can supply values for these parameters in one of the following ways:

- You can specify one or more host variables.
- You can specify a system-descriptor area.
- You can specify a pointer to an **sqlda** structure.

For more information, see PREPARE statement on page 623.

If you know the number and the order of parameters to be supplied at runtime and their data types, you can define the parameters that are needed by the statement as host variables in your program. You pass parameters to the database server positionally, by opening the cursor with the USING keyword, followed by the names of the variables in their sequential order. These variables are matched with the SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement question-mark (?) placeholders in a one-to-one correspondence, from left to right.

You cannot include indicator variables of ESQL/C in the list of variables. To use an indicator variable, you must include the SELECT or EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement text as part of the DECLARE statement, rather than the identifier of a prepared statement.

You must supply one host variable name for each placeholder. The data type of each variable must be compatible with the corresponding type that the prepared statement requires. The following code fragment opens a Select cursor and specifies host variables in the USING clause:

```
sprintf (select_1, "%s %s %s %s %s",
   "SELECT o.order_num, sum(total price)",
   "FROM orders o, items i",
   "WHERE o.order_date > ? AND o.customer_num = ?",
   "AND o.order_num = i.order_num",
   "GROUP BY o.order_num");
EXEC SQL prepare statement_1 from :select_1;
EXEC SQL declare q_curs cursor for statement_1;
EXEC SQL open q_curs using :o_date, :o.customer_num;
```

The following example illustrates the USING clause of the OPEN statement with an EXECUTE FUNCTION statement in the code fragment:

```
stcopy ("EXECUTE FUNCTION one_func(?, ?)", exfunc_stmt);
EXEC SQL prepare exfunc_id from :exfunc_stmt;
EXEC SQL declare func_curs cursor for exfunc_id;
EXEC SQL open func_curs using :arg1, :arg2;
```

Specifying a System Descriptor Area (ESQL/C)

If you do not know the number of parameters to be supplied at runtime or their data types, you can associate input values from a system-descriptor area. A system-descriptor area describes the data type and memory location of one or more values to replace question-mark (?) placeholders.

A system-descriptor area conforms to the X/Open standards.

Use the SQL DESCRIPTOR keywords to introduce the name of a system descriptor area as the location of the parameters.

The COUNT field in the system-descriptor area corresponds to the number of dynamic parameters in the prepared statement. The value of COUNT must be less than or equal to the number of item descriptors that were specified when the system-descriptor area was allocated with the ALLOCATE DESCRIPTOR statement. You can obtain the value of a field with the GET DESCRIPTOR statement and set the value with the SET DESCRIPTOR statement.

The following example shows the OPEN ... USING SQL DESCRIPTOR statement:

```
EXEC SQL allocate descriptor 'desc1';
...
EXEC SQL open selcurs using sql descriptor 'desc1';
```

As the example indicates, the system descriptor area must be allocated before you reference it in the OPEN statement.

Specifying a Pointer to an sqlda Structure (ESQL/C)

If you do not know the number of parameters to be supplied at runtime, or their data types, you can associate input values from an **sqlda** structure. An **sqlda** structure lists the data type and memory location of one or more values to replace question-mark (?) placeholders.

Use the DESCRIPTOR keyword to introduce a pointer to the **sqlda** structure as the location of the parameters.

The **sqlda** value specifies the number of input values that are described in occurrences of **sqlvar**. This number must correspond to the number of dynamic parameters in the prepared statement.

Example of Specifying a Pointer to an sqlda Structure

The following example shows an OPEN ... USING DESCRIPTOR statement:

```
struct sqlda *sdp;
...
EXEC SQL open selcurs using descriptor sdp;
```

Using the WITH REOPTIMIZATION Option (ESQL/C)

Use the WITH REOPTIMIZATION keywords to reoptimize your query plan. When you prepare SELECT, EXECUTE FUNCTION, or EXECUTE PROCEDURE statements, the database server uses a query plan to optimize the query. If you later modify the data associated with the prepared statement, you can compromise the effectiveness of the query plan for that statement. In other words, if you change the data, you might deoptimize your query. To ensure optimization of your query, you can prepare the statement again, or open the cursor again using the WITH REOPTIMIZATION option.

You should generally use the WITH REOPTIMIZATION option, because it provides the following advantages over preparing a statement again:

- Rebuilds only the query plan, rather than the entire statement
- Uses fewer resources
- Reduces overhead
- Requires less time

The WITH REOPTIMIZATION option forces the database server to optimize the query-design plan before it processes the OPEN cursor statement.

The following examples use the WITH REOPTIMIZATION keywords:

```
EXEC SQL open selcurs using descriptor sdp with reoptimization;
```

Relationship Between OPEN and FREE

The database server allocates resources to prepared statements and open cursors. If you execute a FREE *statement_id* or FREE *statement_id_var* statement, you can still open the cursor associated with the freed statement ID. If you release resources with a FREE *cursor_id* or FREE *cursor_id_var* statement, however, you cannot use the cursor unless you declare the cursor again.

Similarly, if you use the SET AUTOFREE statement for one or more cursors, when the program closes the specific cursor, the database server automatically frees the cursor-related resources. In this case, you cannot use the cursor unless you declare the cursor again.

DDL Operations on Tables Referenced by Cursors

Various DDL statements can drop, rename, or alter the schema of a table that is referenced directly (or indirectly, by the identifier of a prepared statement) in the DECLARE statement that defines a cursor. Subsequent OPEN operations on the cursor might fail with error -710, or might produce unexpected results. Changing the number of columns or the data type of a column has this effect, and the user typically must reissue the DESCRIBE statement, the PREPARE statement, and (for cursors associated with routines) the UPDATE STATISTICS statement for any SPL routines that reference a table whose schema has been modified.

These restrictions do not apply, however, if an index is added or dropped when automatic recompilation is enabled for prepared objects and for SPL routines that reference tables that ALTER TABLE, CREATE INDEX, or DROP INDEX operations have modified. This is the default behavior of Informix®. For more information about enabling or disabling automatic recompilation after schema changes, see the description of the IFX_AUTO_REPREPARE option to the SET ENVIRONMENT statement. For more information about the AUTO_REPREPARE configuration parameter, see your *HCL® Informix® Administrator's Reference*.

When the AUTO_REPREPARE configuration parameter and the IFX_AUTO_REPREPARE session environment variable are set to disable recompilation of prepared objects, however, adding an index to a table that is referenced directly or indirectly in a DECLARE statement can similarly invalidate the associated cursor. Subsequent OPEN statements that specify the invalid cursor fail, even if they include the WITH REOPTIMIZATION keywords. If an index is added to the table that is associated with a cursor while automatic recompilation is disabled, you must prepare the statement again and declare the cursor again before you can open the cursor. For cursors associated with calls to SPL routines, you must run the UPDATE STATISTICS statement for routines that reference tables to which an index has been added or dropped. You cannot simply reopen a cursor that is based on a prepared statement that is no longer valid.

OUTPUT statement

Use the OUTPUT statement to send the results of a query to an operating-system file or to a program.

Syntax



Element	Description	Restrictions	Syntax
filename	Path and filename where query results are written. The default path is the current directory.	Can specify a new or existing file. If the file exists, query results overwrite the current contents of the file.	Must conform to the rules of your operating system.
program	Name of a program to receive the query results as input	Program must exist, must be known to the operating system, and must be able to read the results of a query.	Must conform to the rules of your operating system.

Usage

The OUTPUT statement writes query results in an operating-system file, or pipes query results to another program. You can optionally specify whether column headings are omitted from the query output. This statement is an extension to the ANSI/ISO standard for SQL. You can use this statement only with .

Sending Query Results to a File

To send the results of a query to an operating-system file, specify the full pathname for the file. If the file already exists, the output overwrites the current contents.

The following examples show how to send the result of a query to an operating-system file. The example uses UNIX™ file naming conventions.

```
OUTPUT TO /usr/april/query1
   SELECT * FROM cust_calls WHERE call_code = 'L'
```

Displaying Query Results Without Column Headings

About this task

To display the results of a query without column headings, use the WITHOUT HEADINGS keywords.

Sending Query Results to Another Program

You can use the keyword PIPE to send the query results to another program, as the following example shows:

```
OUTPUT TO PIPE more
   SELECT customer_num, call_dtime, call_code
      FROM cust_calls;
```

PREPARE statement

Use the PREPARE statement to parse, validate, and generate an execution plan for one or more SQL statements at runtime.

Syntax

(explicit id unique_636_Connect_42_prep001) unique_636_Connect_42_prep001(explicit id unique_636_Connect_42_prep002) unique_636_Connect_42_prep002

PREPARE { *statement_id* | [] *statement_id_var* } **FROM** { *' statement_text '* | [] *statement_var* | [] *char_expression* }

Element	Description	Restrictions	Syntax
char_expression	Expression that evaluates to the text of a single SQL statement	Statement must be a SELECT, EXECUTE FUNCTION, or EXECUTE PROCEDURE	Expression on page 1003
statement_id	Identifier declared here for the prepared object	Must be unique in the routine among names of cursors and prepared objects (and in SPL, among variables)	Identifier on page 1231
statement_id_var	Host variable storing *statement_id*	Must have been previously declared as a character data type	Language specific
statement_text	Text of the SQL statement(s) to prepare	See #unique_886 and Statement Text on page 626.	Quoted String on page 1202.
statement_var	Host variable storing the text of one or more SQL statements	Must be a character data type. Not valid if the SQL statement(s) contains the Collection-Derived Table segment.	Language specific

Usage

Use this statement in ESQL/C or SPL routines.

The PREPARE statement enables your program to assemble the text of one (or for ESQL/C, more than one) SQL statement at runtime, to declare an identifier for the resulting *prepared object*, and to make it executable. This dynamic form of SQL is accomplished in three steps:

1. The PREPARE statement accepts statement text as input, either as a quoted string, or an ESQL/C character variable, or (in SPL) as the value to which a character expression evaluates. Statement text can contain question-mark (?) placeholders to represent values that are to be defined when the statement is executed.
2. The OPEN statement (and in ESQL/C routines, the EXECUTE statement) can supply the required input values and execute the prepared statement once or many times.
3. Resources allocated to the prepared statement can be released later using the FREE statement.

For more information about the replacement of placeholders in prepared statements with runtime values, see the section Preparing Statements That Receive Parameters on page 630.

The same collating order that is current when you create a prepared object is also used when that object is executed, even if the execution-time collation of the session (or of **DB_LOCALE**) is different.

Restrictions

The number of prepared objects in a single program is limited by available memory. These include statement identifiers declared in PREPARE statements (*statement_id* or *statement_id_var*) and declared cursors. To avoid exceeding the limit, use the FREE statement to release some statements or cursors.

In SPL routines, a prepared object can include the text of no more than one SQL statement, and that statement must be either an EXECUTE FUNCTION, EXECUTE PROCEDURE, or SELECT statement, but the SELECT statement cannot include the INTO *variable*, INTO TEMP, or FOR UPDATE clause.

An expression that specifies the statement text in an SPL routine must evaluate to a CHAR, LVARCHAR, NCHAR, NVARCHAR, or VARCHAR data type. You must explicitly cast to one of these types an expression of any other text data type, such as a UDT.

For restrictions in ESQL/C routines on the SQL statements in the character string, see Restricted Statements in Single-Statement Prepares on page 628 and Restricted Statements in Multistatement Prepared Objects.

Declaring a Statement Identifier

PREPARE sends the statement text to the database server, which analyzes the statement text. If the text contains no syntax errors, the database server translates it to an internal form. This translated statement is saved for later execution in a data structure that the PREPARE statement allocates. The name of the structure is the value that is assigned to the statement identifier in the PREPARE statement. Subsequent SQL statements can refer to the structure by using the same statement identifier that was used in the PREPARE statement.

A subsequent FREE statement releases the database server resources that were allocated to the statement. After you release these resources with FREE, you cannot use the statement identifier in a DECLARE statement or (in ESQL/C) with the EXECUTE statement until you prepare the statement again.

The database server resources for the prepared objects that an SPL routine defines are released automatically when the routine exits.

Scope of Statement Identifiers

An ESQL/C program can consist of one or more source-code files. By default, the scope of reference of a statement identifier is global to the program. Therefore, a statement identifier that is prepared in one file can be referenced from another file.

In a multiple-file program, if you want to limit the scope of reference of a statement identifier to the file in which it is prepared, preprocess all the files with the **-local** command-line option.

Releasing a Statement Identifier

A statement identifier can represent only one SQL statement or (in ESQL/C) one semicolon-separated list of SQL statements at a time. A new PREPARE statement can specify an existing statement identifier if you want to bind the identifier to a different SQL statement text.

The PREPARE statement supports dynamic statement-identifier names, which allow you to prepare a statement identifier as an identifier or (in ESQL/C) as a host variable of a data type that can contain a character string. The first example that follows shows a statement identifier that was specified as a host variable. The second specifies a statement identifier as a character string.

```
stcopy ("query2", stmtid);
EXEC SQL prepare :stmtid from 'select * from customer';
```

```
EXEC SQL prepare query2 from 'select * from customer';
```

The variable must be a character data type. In C, it must be declared as **char**.

In an SPL routine, statement identifiers that the PREPARE statement declares are automatically defined in the local scope. Do not attempt to declare a statement identifier as having local or global scope. A statement identifier defined in one SPL routine is not visible to any other SPL routine that the same session calls. SPL statement identifiers share the same namespace as SPL variables and cursor names.

Statement Text

Statement text can be specified in the PREPARE statement

- as a quoted string
- or as text that is stored in an ESQL/C program variable
- or (in SPL routines) as a character expression.

The following restrictions apply to the statement text:

- The text can contain only SQL statements. It cannot contain statements or comments from the host programming language.
- The text can contain comments preceded by a double hyphen (--), or that are enclosed in braces ({ }) or in C-style slash and asterisk (/* */) delimiters.

  These symbols introduce or enclose SQL comments. For more information on SQL comment symbols, see How to Enter SQL Comments on page 6.
- The text can contain either a single SQL statement or (in ESQL/C routines) a series of statements that are separated by semicolon (;) symbols.

  For a list of SQL statements that cannot be prepared, see Restricted Statements in Single-Statement Prepares on page 628. For more information on how to prepare multiple SQL statements, see #unique_886.
- The text cannot include an embedded SQL statement prefix or terminator, such as a dollar sign ($) or the words EXEC SQL.
- Host-language variables are not recognized as such in prepared text.

  Therefore, you cannot prepare a SELECT (or EXECUTE FUNCTION or EXECUTE PROCEDURE) statement that includes an INTO clause, because the INTO clause requires a host-language variable.
- The only identifiers that you can use are names that are defined in the database, such as names of tables and columns. For more information on how to use identifiers in statement text, see Preparing Statements with SQL Identifiers on page 631.
- Use a question mark (?) as a placeholder to indicate where data is supplied when the statement executes, as in this example:

```
EXEC SQL prepare new_cust from
       'insert into customer(fname,lname) values(?,?)';
```

For more information on how to use question marks as placeholders, see Preparing Statements That Receive Parameters on page 630.

If the prepared statement contains the Collection-Derived Table segment or the collection variable, some additional limitations exist on how you can assemble the text for the PREPARE statement. For information about dynamic SQL, see the *HCL® Informix® Enterprise Replication Guide*. SPL routines cannot use dynamic SQL statements to process prepared statements that contain the Collection-Derived Table segment.

Example of a PREPARE statement in an SPL routine

The HCL Informix® SPL language supports single-statement prepared objects.

For example, the following SQL and SPL statements perform these tasks:

1. Create the **cities** table.
2. Populates the **cities** table with four rows of data.
3. Creates the **order_city** SPL routine that defines a prepared statement and a cursor to query the **cities** table:

```
CREATE TABLE cities    -- defines a table
(
  id INT,
  city_name CHAR(50)
);

INSERT INTO cities VALUES (1, 'Chicago');
INSERT INTO cities VALUES (2, 'New York');
INSERT INTO cities VALUES (3, 'San Francisco');
INSERT INTO cities VALUES (4, 'Atlanta');

UPDATE STATISTICS HIGH;

CREATE PROCEDURE order_city() -- defines a UDR
RETURNING INT, CHAR(50);
DEFINE c_num INT;
DEFINE c_name CHAR(50);
DEFINE c_query VARCHAR(250);
LET c_query =
 "SELECT id, city_name FROM cities ORDER BY city_name;";

PREPARE c_stmt FROM c_query;
DECLARE c_cur CURSOR FOR c_stmt;

OPEN c_cur ;
while (1 = 1)
  FETCH c_cur INTO c_num, c_name;
  IF (SQLCODE != 100) THEN
    RETURN c_num, c_name WITH RESUME;
  ELSE
    EXIT;
  END IF
```

```
END WHILE

CLOSE c_cur;
FREE c_cur;
FREE c_stmt;

END PROCEDURE;
```

The following SQL statement invokes the **order_city** routine:

```
EXECUTE PROCEDURE order_city();
```

If the **order_city** function is called from the **dbaccess** utility, this output is displayed:

```
(expression) (expression)

          4 Atlanta
          1 Chicago
          2 New York
          3 San Francisco
```

For an overview with detailed examples of how to create and use prepared objects and Dynamic SQL in SPL routines, see this IBM® developerWorks® article: Dynamic SQL support in Informix® Dynamic Server Stored Procedure Language

Preparing and Executing User-Defined Routines

The way to prepare a user-defined routine (UDR) depends on whether the UDR is a user-defined procedure or a user-defined function:

- To prepare a user-defined procedure, prepare the EXECUTE PROCEDURE statement that executes the procedure.

  To execute the prepared procedure, use the EXECUTE statement.
- To prepare a user-defined function, prepare the EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement that executes the function.

  You cannot include the INTO clause of EXECUTE FUNCTION (or EXECUTE PROCEDURE) in the PREPARE statement.

How to execute a prepared user-defined function depends on whether it returns only one group or multiple groups of values. Use the EXECUTE statement for user-defined functions that return only one group of values.

To execute user-defined functions that return more than one group of return values, you must associate the EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement with a cursor.

Restricted Statements in Single-Statement Prepares

In general, you can prepare any data manipulation language (DML) statement.

In Informix®, you can prepare any single SQL statement except for the following statements:

- ALLOCATE COLLECTION
- ALLOCATE DESCRIPTOR

- ALLOCATE ROW
- CLOSE
- CONNECT
- CREATE FUNCTION FROM
- CREATE PROCEDURE FROM
- CREATE ROUTINE FROM
- DEALLOCATE COLLECTION
- DEALLOCATE DESCRIPTOR
- DEALLOCATE ROW
- DECLARE
- DESCRIBE
- DISCONNECT
- EXECUTE
- EXECUTE IMMEDIATE
- FETCH
- FLUSH
- FREE
- GET DESCRIPTOR
- GET DIAGNOSTICS
- INFO
- LOAD
- OPEN
- OUTPUT
- PREPARE
- PUT
- SET AUTOFREE
- SET CONNECTION
- SET DEFERRED_PREPARE
- SET DESCRIPTOR
- UNLOAD
- WHENEVER

You can prepare a SELECT statement. If SELECT includes the INTO TEMP clause, an ESQL/C program can execute the prepared statement with an EXECUTE statement. If it does not include the INTO TEMP clause, the statement returns rows of data. Use DECLARE, OPEN, and FETCH cursor statements to retrieve the rows.

In ESQL/C, a prepared SELECT statement can include a FOR UPDATE clause. This clause is used with the DECLARE statement to create an update cursor. The next example shows a SELECT statement with a FOR UPDATE clause in :

```
sprintf(up_query, "%s %s %s",
   "select * from customer ",
   "where customer_num between ? and ? ",
   "for update");
EXEC SQL prepare up_sel from :up_query;
```

```
EXEC SQL declare up_curs cursor for up_sel;
EXEC SQL open up_curs using :low_cust,:high_cust;
```

Preparing Statements When Parameters Are Known

In some prepared statements, all necessary information is known at the time the statement is prepared. The following example in shows two statements that were prepared from constant data:

```
sprintf(redo_st, "%s %s",
   "drop table workt1; ",
   "create table workt1 (wtk serial, wtv float)" );
EXEC SQL prepare redotab from :redo_st;
```

Preparing Statements That Receive Parameters

In some statements, parameters are unknown when the statement is prepared because a different value can be inserted each time the statement is executed. In these statements, you can use a question-mark (?) placeholder where a parameter must be supplied when the statement is executed.

The PREPARE statements in the following examples show some uses of question-mark (?) placeholders:

```
EXEC SQL prepare s3 from
   'select * from customer where state matches ?';
EXEC SQL prepare in1 from 'insert into manufact values (?,?,?)';
sprintf(up_query, "%s %s",
   "update customer set zipcode = ?"
   "where current of zip_cursor");
EXEC SQL prepare update2 from :up_query;
EXEC SQL prepare exfunc from
   'execute function func1 (?, ?)';
```

You can use a placeholder to defer evaluation of a value until runtime only for an expression, but not for an SQL identifier, except as noted in Preparing Statements with SQL Identifiers on page 631.

The following example of the code fragment prepares a statement from a variable that is named **demoquery**. The text in the variable includes one question-mark (?) placeholder. The prepared statement is associated with a cursor and, when the cursor is opened, the USING clause of the OPEN statement supplies a value for the placeholder:

```
EXEC SQL BEGIN DECLARE SECTION;
   char queryvalue [6];
   char demoquery  [80];
EXEC SQL END DECLARE SECTION;

EXEC SQL connect to 'stores_demo';
sprintf(demoquery, "%s %s",
      "select fname, lname from customer ",
      "where lname > ? ");
EXEC SQL prepare quid from :demoquery;
EXEC SQL declare democursor cursor for quid;
stcopy("C", queryvalue);
EXEC SQL open democursor using :queryvalue;
```

The USING clause is available in both OPEN statements that are associated with a cursor and EXECUTE statements (all other prepared statements).

You can use a question-mark (?) placeholder to represent the name of the or SPL collection variable.

Preparing Statements with SQL Identifiers

In general, you must specify SQL identifiers explicitly in the statement text when you prepare the statement. In a few special cases, however, you can use the question-mark (?) placeholder for an SQL identifier:

- For the database name in the DATABASE statement.
- For the dbspace name in the IN *dbspace* clause of the CREATE DATABASE statement.
- For the cursor name in statements that use cursor names.

Obtaining SQL Identifiers from User Input

If a prepared statement requires identifiers, but the identifiers are unknown when you write the prepared statement, you can construct a statement that receives SQL identifiers from user input.

The following example prompts the user for the name of a table and uses that name in a SELECT statement. Because this name is unknown until runtime, the number and data types of the table columns are also unknown. Therefore, the program cannot allocate host variables to receive data from each row in advance. Instead, this program fragment describes the statement into an **sqlda** descriptor and fetches each row with the descriptor. The fetch puts each row into memory locations that the program provides dynamically.

If a program retrieves all the rows in the active set, the FETCH statement would be placed in a loop that fetched each row. If the FETCH statement retrieves more than one data value (column), another loop exists after the FETCH, which performs some action on each data value:

```
#include <stdio.h>
EXEC SQL include sqlda;
EXEC SQL include sqltypes;
char *malloc( );

main()
{
   struct sqlda *demodesc;
   char tablename[19];
   int i;
EXEC SQL BEGIN DECLARE SECTION;
   char demoselect[200];
EXEC SQL END DECLARE SECTION;

/*  This program selects all the columns of a given tablename.
       The tablename is supplied interactively. */

EXEC SQL connect to 'stores_demo';
printf( "This program does a select * on a table\n" );
printf( "Enter table name: " );
scanf( "%s", tablename );
sprintf(demoselect, "select * from %s", tablename );

EXEC SQL prepare iid from :demoselect;
EXEC SQL describe iid into demodesc;
```

```
/* Print what describe returns */

for ( i = 0;  i < demodesc->sqld; i++ )
   prsqlda (demodesc->sqlvar + i);
```

```
/* Assign the data pointers. */

for ( i = 0;  i < demodesc->sqld; i++ )
   {
   switch (demodesc->sqlvar[i].sqltype & SQLTYPE)
      {
      case SQLCHAR:
         demodesc->sqlvar[i].sqltype = CCHARTYPE;
         /* make room for null terminator */
         demodesc->sqlvar[i].sqllen++;
         demodesc->sqlvar[i].sqldata =
            malloc( demodesc->sqlvar[i].sqllen );
         break;

      case SQLSMINT:    /* fall through */
      case SQLINT:      /* fall through */
      case SQLSERIAL:
         demodesc->sqlvar[i].sqltype = CINTTYPE;
         demodesc->sqlvar[i].sqldata =
            malloc( sizeof( int ) );
         break;
      /*  And so on for each type.  */
      }
   }

/* Declare and open cursor for select . */
EXEC SQL declare d_curs cursor for iid;
EXEC SQL open d_curs;

/* Fetch selected rows one at a time into demodesc. */
for( ; ; )
   {
   printf( "\n" );
   EXEC SQL fetch d_curs using descriptor demodesc;
   if ( sqlca.sqlcode != 0 )
       break;
   for ( i = 0;  i < demodesc->sqld; i++ )
      {
      switch (demodesc->sqlvar[i].sqltype)
         {
         case CCHARTYPE:
            printf( "%s: \"%s\n", demodesc->sqlvar[i].sqlname,
               demodesc->sqlvar[i].sqldata );
            break;
         case CINTTYPE:
            printf( "%s: %d\n", demodesc->sqlvar[i].sqlname,
               *((int *) demodesc->sqlvar[i].sqldata) );
            break;
         /* And so forth for each type... */
            }
      }
   }
```

```
EXEC SQL close d_curs;
EXEC SQL free d_curs;
/*  Free the data memory.  */

for ( i = 0;  i < demodesc->sqld; i++ )
   free( demodesc->sqlvar[i].sqldata );
free( demodesc );

printf ("Program Over.\n");
}

prsqlda(sp)
   struct sqlvar_struct *sp;

   {
   printf ("type = %d\n", sp->sqltype);
   printf ("len = %d\n", sp->sqllen);
   printf ("data = %lx\n", sp->sqldata);
   printf ("ind = %lx\n", sp->sqlind);
   printf ("name = %s\n", sp->sqlname);
   }
```

Using Prepared Statements for Efficiency

To increase performance efficiency, you can use the PREPARE statement and an EXECUTE statement in a loop to eliminate overhead that redundant parsing and optimizing cause. For example, an UPDATE statement that is located within a WHILE loop is parsed each time the loop runs. If you prepare the UPDATE statement outside the loop, the statement is parsed only once, eliminating overhead and speeding statement execution. The following example shows how to prepare statements to improve performance:

```
EXEC SQL BEGIN DECLARE SECTION;
   char disc_up[80];
   int cust_num;
EXEC SQL END DECLARE SECTION;
main()
{
   sprintf(disc_up, "%s %s","update customer ",
      "set discount = 0.1 where customer_num = ?");
   EXEC SQL prepare up1 from :disc_up;
   while (1)
      {
      printf("Enter customer number (or 0 to quit): ");
      scanf("%d", cust_num);
      if (cust_num == 0)
         break;
      EXEC SQL execute up1 using :cust_num;
      }
}
```

Like the SQL statement cache, prepared statements can reduce how often the same query plan is reoptimized, thereby conserving resources in some contexts. The section Prepared Statements and the Statement Cache on page 836 discusses the use of prepared DML statements, cursors, and the SQL statement cache as combined or alternative techniques for improving query performance.

DDL Operations on Tables Referenced in Prepared Objects

Various DDL statements can drop, rename, or alter the schema of a table that a prepared object references, but subsequent attempts to execute the prepared object might fail with error -710, or might produce unexpected results.

These restrictions do not necessarily apply, however, if an index is added or dropped when automatic recompilation is enabled for prepared objects and routines that directly reference tables that ALTER TABLE, CREATE INDEX, or DROP INDEX operations have modified. This is the default behavior of Informix®. For more information about using the SET ENVIRONMENT IFX_AUTO_REPREPARE statement to enable or disable automatic recompilation after changes to the schema of a table, and for contexts where the database server issues error -710 even when automatic recompilation is enabled, see IFX_AUTO_REPREPARE session environment option.

When the AUTO_REPREPARE configuration parameter and the IFX_AUTO_REPREPARE session environment variable are set to disable automatic recompilation, however, adding an index to a table that a prepared statement references indirectly can similarly invalidate the prepared statement. A subsequent OPEN statement fails if the cursor refers to the invalid prepared statement, even if the OPEN statement includes the WITH REOPTIMIZATION keywords. If an index on an indirectly referenced table is added after the statement was prepared while automatic recompilation is disabled, you must prepare the statement again and declare the cursor again. You cannot simply reopen the cursor if it is based on a prepared statement that is no longer valid.

Related Statements

Related statements: CLOSE statement on page 170, DECLARE statement on page 433, DESCRIBE statement on page 459, EXECUTE statement on page 496, FREE statement on page 524, OPEN statement on page 616, SET AUTOFREE statement on page 773, and SET DEFERRED_PREPARE statement on page 799

For information about basic concepts that relate to the PREPARE statement, see the *HCL® Informix® Guide to SQL: Tutorial*.

For information about more advanced concepts that relate to the PREPARE statement, see the *HCL® Informix® Enterprise Replication Guide*.

PUT statement

Use the PUT statement to store a row in an insert buffer for later insertion into the database.

Syntax

PUT { *cursor_id_var* | *cursor_id* } [{ **FROM** *output_var* [{ **INDICATOR***indicator_var* | $*indicator_var* | :*indicator_var* }] | **USING** { **SQL DESCRIPTOR** { *'descriptor '* | *descriptor_var* } | **DESCRIPTOR***sqlda_pointer* } }]

Element	Description	Restrictions	Syntax
cursor_id	Name of a cursor	Must be open	Identifier on page 1231
cursor_id_var	Host variable = *cursor_id*	Must be a character type; cursor must be open	Language specific

Element	Description	Restrictions	Syntax
descriptor	Name of a system-descriptor area	Must already be allocated	Quoted String on page 1202
descriptor_var	Host-variable that contains *descriptor*	Must already be allocated	Quoted String on page 1202
indicator_var	Host variable to receive a return code if corresponding *output_var* receives a NULL value	Cannot be a DATETIME or INTERVAL data type	Language specific
output_var	Host variable whose contents replace a question-mark (?) placeholder in a prepared INSERT statement	Must be a character data type	Language specific
sqlda_pointer	Pointer to an **sqlda** structure	First character cannot be the ($) or (:) symbol	DESCRIBE statement on page 459

Usage

This statement is an extension to the ANSI/ISO standard for SQL. You can use this statement with ESQL/C.

PUT stores a row in an *insert buffer* that is created when the cursor is opened.

If the buffer has no room for the new row when the statement executes, the buffered rows are written to the database in a block, and the buffer is emptied. As a result, some PUT statement executions cause rows to be written to the database, and some do not. You can use the FLUSH statement to write buffered rows to the database without adding a new row. The CLOSE statement writes any remaining rows before it closes an Insert cursor.

If the current database uses explicit transactions, you must execute a PUT statement within a transaction.

The following example uses a PUT statement in :

```
EXEC SQL prepare ins_mcode from
   'insert into manufact values(?,?)';
EXEC SQL declare mcode cursor for ins_mcode;
EXEC SQL open mcode;
EXEC SQL put mcode from :the_code, :the_name;
```

The PUT statement is not an X/Open SQL statement. Therefore, you get a warning message if you compile a PUT statement in X/Open mode.

Supplying Inserted Values

The values in the inserted row can come from one of the following sources:

- Constant values that are written into the INSERT statement
- Program variables that are named in the INSERT statement

- Program variables in the FROM clause of the PUT statement
- Values that are prepared in memory addressed by an **sqlda** structure or a system-descriptor area and then specified in the USING clause of the PUT statement

The system descriptor area or **sqlda** structure that *descriptor* or *sqlda_pointer* references must define a data type and memory location of each value that corresponds to a question-mark (?) placeholder in a prepared INSERT statement.

Using Constant Values in INSERT

The VALUES clause lists the values for the inserted columns. One or more of these values can be constants (that is, numbers or character strings).

When *all* the inserted values are constants, the PUT statement has a special effect. Instead of creating a row and putting it in the buffer, the PUT statement merely increments a counter. When you use a FLUSH or CLOSE statement to empty the buffer, one row and a repetition count are sent to the database server, which inserts that number of rows. In the following example, 99 empty customer records are inserted into the **customer** table. Because all values are constants, no disk output occurs until the cursor closes. (The constant zero for **customer_num** causes generation of a SERIAL value.) The following example inserts 99 empty customer records into the customer table:

```
int count;
EXEC SQL declare fill_c cursor for
   insert into customer(customer_num) values(0);
EXEC SQL open fill_c;
for (count = 1; count <= 99; ++count)
   EXEC SQL put fill_c;
EXEC SQL close fill_c;
```

Naming Program Variables in INSERT

When you associate the INSERT statement with a cursor (in the DECLARE statement), you create an Insert cursor. In the INSERT statement, you can name program variables in the VALUES clause. When each PUT statement is executed, the contents of the program variables at that time are used to populate the row that is inserted into the buffer.

If you are creating an Insert cursor (using DECLARE with INSERT), you must use only program variables in the VALUES clause. Variable names are not recognized in the context of a prepared statement; you associate a prepared statement with a cursor through its statement identifier.

The following example illustrates the use of an Insert cursor. The code includes the following statements:

- The DECLARE statement associates a cursor called **ins_curs** with an INSERT statement that inserts data into the **customer** table.

  The VALUES clause specifies a data structure that is called **cust_rec**; the preprocessor converts **cust_rec** to a list of values, one for each component of the structure.
- The OPEN statement creates a buffer.
- A user-defined function (not defined within this example) obtains customer information from user input and stores it in **cust_rec**.

- The PUT statement composes a row from the current contents of the **cust_rec** structure and sends it to the row buffer.
- The CLOSE statement inserts into the **customer** table any rows that remain in the row buffer and closes the Insert cursor:

```
int keep_going = 1;
EXEC SQL BEGIN DECLARE SECTION
   struct cust_row { /* fields of a row of customer table */ } cust_rec;
EXEC SQL END DECLARE SECTION
EXEC SQL declare ins_curs cursor for
      insert into customer values (:cust_row);
EXEC SQL open ins_curs;
while ( (sqlca.sqlcode == 0) && (keep_going) )
```

```
  {
keep_going = get_user_input(cust_rec); /* ask user for new customer */
   if (keep_going )                    /* user did supply customer info
*/
      {
      cust_rec.customer_num = 0;       /* request new serial value */
      EXEC SQL put ins_curs;
      }
   if (sqlca.sqlcode == 0)             /* no error from PUT */
      keep_going = (prompt_for_y_or_n("another new customer") =='Y')
   }
EXEC SQL close ins_curs;
```

Use an indicator variable if the data to be inserted might be NULL.

Naming Program Variables in PUT

When the INSERT statement is prepared (see PREPARE statement on page 623), you cannot use program variables in its VALUES clause, but you can represent values by a question-mark (?) placeholder. List the program variables in the FROM clause of the PUT statement to supply the missing values.

The following example lists host variables in a PUT statement:

```
char answer [1] = 'y';
EXEC SQL BEGIN DECLARE SECTION;
   char ins_comp[80];
   char u_company[20];
EXEC SQL END DECLARE SECTION;

main()
{
   EXEC SQL connect to 'stores_demo';
   EXEC SQL prepare ins_comp from
      'insert into customer (customer_num, company) values (0, ?)';
   EXEC SQL declare ins_curs cursor for ins_comp;
   EXEC SQL open ins_curs;

   while (1)
      {
```

```
      printf("\nEnter a customer: ");
       gets(u_company);
      EXEC SQL put ins_curs from :u_company;
      printf("Enter another customer (y/n) ? ");
      if (answer = getch() != 'y')
         break;
      }
   EXEC SQL close ins_curs;
   EXEC SQL disconnect all;
}
```

Indicator variables are optional, but you should use an indicator variable if the possibility exists that *output_var* might contain a NULL value. If you specify the indicator variable without the INDICATOR keyword, you cannot put a blank space between *output_var* and *indicator_var*.

Using the USING Clause

If you do not know the number of parameters to be supplied at runtime or their data types, you can associate input values from a system-descriptor area or an **sqlda** structure. Both of these descriptor structures describe the data type and memory location of one or more values to replace question-mark (?) placeholders.

Each time the PUT statement executes, the values that the descriptor structure describes are used to replace question-mark (?) placeholders in the INSERT statement. This process is similar to using a FROM clause with a list of variables, except that your program has full control over the memory location of the data values.

Specifying a System-Descriptor Area

The SQL DESCRIPTOR option specifies the name of a system-descriptor area.

The **COUNT** field in the system-descriptor area corresponds to the number of dynamic parameters in the prepared statement. The value of **COUNT** must be less than or equal to the number of item descriptors that were specified when the system-descriptor area was allocated with the ALLOCATE DESCRIPTOR statement. You can obtain the value of a field with the GET DESCRIPTOR statement and set the value with the SET DESCRIPTOR statement.

A system-descriptor area conforms to the X/Open standards.

The following example shows how to associate values from a system-descriptor area:

```
EXEC SQL allocate descriptor 'desc1';
...
EXEC SQL put selcurs using sql descriptor 'desc1';
```

Specifying an sqlda Structure

Use the **DESCRIPTOR** option to introduce the name of a pointer to an **sqlda** structure. The following example shows how to associate values from an **sqlda** structure:

```
EXEC SQL put selcurs using descriptor pointer2;
```

Inserting into a Collection Cursor

A Collection cursor allows you to access the individual elements of a collection variable. To declare a Collection cursor, use the DECLARE statement and include the Collection-Derived Table segment in the INSERT statement that you associate with the cursor. Once you open the Collection cursor with the OPEN statement, the cursor can put elements in the collection variable.

To put elements, one at a time, into the Insert cursor, use the PUT statement and the FROM clause. The PUT statement identifies the Collection cursor that is associated with the collection variable. The FROM clause identifies the element value to be inserted into the cursor. The data type of any host variable in the FROM clause must match the element type of the collection.

Important: The collection variable stores the elements of the collection. However, it has no intrinsic connection with a database column. Once the collection variable contains the correct elements, you must then save the variable into the actual collection column with the INSERT or UPDATE statement.

Suppose you have a table called **children** with the following schema:

```
CREATE TABLE children
(
   age      SMALLINT,
   name     VARCHAR(30),
   fav_colors SET(VARCHAR(20) NOT NULL)
);
```

The following program fragment shows how to use an Insert cursor to put elements into a collection variable called **child_colors**:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection child_colors;
   char *favorites[]
   (
      "blue",
      "purple",
      "green",
      "white",
      "gold",
      0
   );
   int a = 0;
   char child_name[21];
EXEC SQL END DECLARE SECTION;

EXEC SQL allocate collection :child_colors;

/* Get structure of fav_colors column for untyped
 * child_colors collection variable */
EXEC SQL select fav_colors into :child_colors
   from children where name = :child_name;
/* Declare insert cursor for child_colors collection
 * variable and open this cursor */
EXEC SQL declare colors_curs cursor for
   insert into table(:child_colors)
```

```
   values (?);
EXEC SQL open colors_curs;
/* Use PUT to gather the favorite-color values
 * into a cursor */
while (fav_colors[a])
```

```
{
   EXEC SQL put colors_curs from :favorites[:a];
   a++
   ...
}
/* Flush cursor contents to collection variable */
EXEC SQL flush colors_curs;
EXEC SQL update children set fav_colors = :child_colors;

EXEC SQL close colors_curs;
EXEC SQL deallocate collection :child_colors;
```

After the FLUSH statement executes, the collection variable, **child_colors**, contains the elements {"`blue", "purple", "green", "white", "gold`"}. The UPDATE statement at the end of this program fragment saves the new collection into the **fav_colors** column of the database. Without this UPDATE statement, the new collection would not be added to the collection column.

Writing Buffered Rows

To open an Insert cursor, the OPEN statement creates an insert buffer. The PUT statement puts a row into this insert buffer. The buffered rows are inserted into the database table as a block only when necessary; this process is called *flushing the buffer*. The buffer is flushed after any of the following events:

- Buffer is too full to hold the new row at the start of a PUT statement.
- A FLUSH statement executes.
- A CLOSE statement closes the cursor.
- An OPEN statement specifies an already open cursor, closing it before reopening it. (This implicit CLOSE statement flushes the buffer.)
- A COMMIT WORK statement executes.
- Buffer contains BYTE or TEXT data (flushed after a single PUT statement).

If the program terminates without closing an Insert cursor, the buffer remains unflushed. Rows that were inserted into the buffer since the last flush are lost. Do not rely on the end of the program to close the cursor and flush the buffer.

Error Checking

The **sqlca** structure contains information on the success of each PUT statement as well as information that lets you count the rows that were inserted. The result of each PUT statement is contained in the following fields of the **sqlca**: **sqlca.sqlcode**, **SQLCODE**, and **sqlca.sqlerrd[2]**.

Data buffering with an Insert cursor means that errors are not discovered until the buffer is flushed. For example, an input value that is incompatible with the data type of the column for which it is intended is discovered only when the buffer is

flushed. When an error is discovered, buffered rows that were not inserted before the error are *not* inserted; they are lost from memory.

The **SQLCODE** field is set to `0` if no error occurs; otherwise, it is set to an error code. The third element of the **sqlerrd** array is set to the number of rows that were successfully inserted into the database:

- If any row is put into the insert buffer, but *not* written to the database, **SQLCODE** and **sqlerrd** are set to `0` (**SQLCODE** because no error occurred, and **sqlerrd** because no rows were inserted).
- If a block of buffered rows is written to the database during the execution of a PUT statement, **SQLCODE** is set to `0` and **sqlerrd** is set to the number of rows that was successfully inserted into the database.
- If an error occurs while the buffered rows are written to the database, **SQLCODE** indicates the error, and **sqlerrd** contains the number of successfully inserted rows. (The uninserted rows are discarded from the buffer.)

Tip: When you encounter an **SQLCODE** error, a **SQLSTATE** error value also exists. See the GET DIAGNOSTICS statement for details of how to obtain the message text.

To count the number of pending and inserted rows in the database

1. Prepare two bigint variables (for example, **total** and **pending)**.
2. When the cursor is opened, set both variables to `0`.
3. Each time a PUT statement executes, increment both **total** and **pending**.
4. Whenever a PUT or FLUSH statement executes or the cursor closes, subtract the third field of the **SQLERRD** array from **pending**.

At any time, (**total** - **pending)** represents the number of rows actually inserted. If no statements fail, **pending** contains zero after the cursor is closed. If an error occurs during a PUT, FLUSH, or CLOSE statement, the value that remains in **pending** is the number of uninserted (discarded) rows.

Related Statements

Related statements: ALLOCATE DESCRIPTOR statement on page 19, CLOSE statement on page 170, DEALLOCATE DESCRIPTOR statement on page 432, FLUSH statement on page 522, DECLARE statement on page 433, #unique_915, OPEN statement on page 616, PREPARE statement on page 623, and #unique_916

For a task-oriented discussion of the PUT statement, see the *HCL® Informix® Guide to SQL: Tutorial*.

For more information about error checking, the system-descriptor area, and the **sqlda** structure, see the *HCL® Informix® Enterprise Replication Guide*.

RELEASE SAVEPOINT statement

Use the RELEASE SAVEPOINT statement to destroy a specified savepoint. The RELEASE SAVEPOINT statement is compliant with the ANSI/ISO standard for SQL.

Syntax

```
RELEASE SAVEPOINT savepoint
```

Element	Description	Restrictions	Syntax
savepoint	Name of the savepoint to be destroyed	Must exist in the current savepoint level	Identifier on page 1231

Usage

Restriction: After this statement executes successfully, rollback to the specified savepoint (or to any other savepoint between the RELEASE SAVEPOINT statement and the specified savepoint) is no longer possible.

The RELEASE SAVEPOINT statement destroys the specified savepoint. Any savepoints set between that savepoint and the RELEASE SAVEPOINT statement in the current savepoint level are also destroyed. Any other savepoints, however, that were set earlier than the specified savepoint in the current savepoint level continue to be active.

The RELEASE SAVEPOINT statement fails with an error in the following contexts:

- No SQL transaction is open.
- No savepoint with the specified name exists in the current savepoint level.
- The statement is part of a triggered action.
- The statement is part of an XA transaction.
- The autocommit transaction mode of the client API is enabled.
- The statement is part of a cross-server distributed SQL transaction in which one of the participating database servers does not support savepoints.
- The statement is issued within a UDR that is invoked within a DML statement.

The identifier of any savepoint that RELEASE SAVEPOINT destroys can be reused in a subsequent SAVEPOINT statement of the same savepoint level, even if the released savepoint was set by a SAVEPOINT statement that included the UNIQUE keyword.

Because savepoints are program objects, not database objects, the RELEASE SAVEPOINT statement has no direct effect on the database or on its system catalog tables. RELEASE SAVEPOINT can affect user tables and the system catalog indirectly, however, if it changes the scope of a subsequent ROLLBACK TO SAVEPOINT operation that cancels uncommitted changes to the database within a different portion of the current savepoint level, as the next example illustrates.

The following program fragment sets and then releases a savepoint called `sp45`:

```
BEGIN WORK;
CREATE DATABASE third_base IN db3 WITH BUFFERED LOG;
SAVEPOINT sp46;
CREATE TABLE tab1 ( col1 INT, col2 CHAR(24));
SAVEPOINT sp45 UNIQUE;
...
CREATE TABLE tab2 ( col1 INT8, col2 LVARCHAR(24000));
SAVEPOINT sp44;
```

```
...
RELEASE SAVEPOINT sp45;
ROLLBACK TO SAVEPOINT;
```

The effect of the RELEASE SAVEPOINT statement in this example is to destroy two savepoints, `sp45` and `sp44`. If the only remaining savepoint in the current savepoint level is `sp46`, the subsequent ROLLBACK TO SAVEPOINT statement cancels the DDL statements that created **tab1** and **tab2**, and any DML operations on those tables that preceded the ROLLBACK statement. The rollback does not, however, cancel the CREATE DATABASE statement that created the **third_base** database. Without the RELEASE SAVEPOINT statement, the CREATE TABLE statement that created **tab1** would not have been cancelled, because Informix® would have treated **sp44** as the default.savepoint that the TO SAVEPOINT clause of the ROLLBACK statement references.

RENAME COLUMN statement

Use the RENAME COLUMN statement to change the name of a column. The RENAME COLUMN statement is an extension to the ANSI/ISO standard for SQL.

Syntax

RENAME COLUMN [*owner* .] *table* . *old_column* **TO** *new_column*

Element	Description	Restrictions	Syntax
new_column	Name that you declare here to replace *old_column*	Must be unique among column names in *table*. See also How Triggers Are Affected on page 644.	Identifier on page 1231
old_column	Column to rename	Must exist within table	Identifier on page 1231
owner	Owner of the table	Must be the owner of table	Owner name on page 1262
table	Table that contains *old_column*	Must be registered in the current database	Identifier on page 1231

Usage

You can rename a column of a table if any of the following conditions are true:

- You own the table or have Alter privilege on the table.
- You have the DBA privilege on the database.

The column can be in a table object that the CREATE EXTERNAL TABLE statement defined.

Example

The following example assigns the new name of **c_num** to the **customer_num** column in the **customer** table:

```
RENAME COLUMN customer.customer_num TO c_num;
```

How Views and Check Constraints Are Affected

If you rename a column that appears in a view, the text of the view definition in the **sysviews** system catalog table is updated to reflect the new column name. If you rename a column that appears in a check constraint, the text of the check constraint in the **syschecks** system catalog table is updated to reflect the new column name.

How Triggers Are Affected

If you rename a column that appears within the definition a trigger, it is replaced with the new name only in the following instances:

- When it appears as part of a correlation name inside the FOR EACH ROW action clause of a trigger
- When it appears as part of a correlation name in the INTO clause of an EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement
- When it appears as a triggering column in the UPDATE clause

When the trigger executes, if the database server encounters a column name that no longer exists in the table, an error is returned.

RENAME CONSTRAINT statement

Use the RENAME CONSTRAINT statement to rename the constraint. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
RENAME CONSTRAINT [ owner . ] old_constraint TO new_constraint
```

Element	Description	Restrictions	Syntax
new_constraint	New name that you declare here for *old_constraint*	Name must be unique to the database	Identifier on page 1231
old_constraint	Constraint name that *new_constraint* replaces	Must exist, but it cannot be a constraint on a system catalog table	Identifier on page 1231
owner	Owner of *old_constraint*	Must be the owner of *old_constraint*	Owner name on page 1262

Usage

You can rename a constraint if any of the following is true:

- You are the owner of the constraint.
- You have the DBA privilege on the database.

When you rename a constraint, the database server changes the constraint name in the **sysconstraints** and **sysobjstate** system catalog tables.

RENAME DATABASE statement

Use the RENAME DATABASE statement to change the name of a database. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
RENAME DATABASE [ owner . ] old_database TO new_database
```

Element	Description	Restrictions	Syntax
new_database	New name that you declare here for *old_database*	Must be unique among database names of the current database server; must not be opened by any user when this statement is issued	Database Name on page 1224
old_database	Name that *new_database* replaces	Must exist on current database server, but it cannot be the name of the current database Cannot be a tenant database.	Database Name on page 1224
owner	Owner of *old_database*	Must be the owner of the database	Owner name on page 1262

Usage

You can rename a database if any of the following is true:

- You created the database.
- You have the DBA privilege on the database.
- The database is not a tenant database. Tenant databases cannot be renamed.

The RENAME DATABASE statement fails with error -9874, however, if the specified database contains any of the following objects:

- a virtual table
- a virtual index
- an R-tree index
- a that references the current name of the database in a user-defined primary access method or in a user-defined secondary access method.

You can only rename databases of the database server to which you are currently connected.

You cannot rename a database from inside an SPL routine.

RENAME INDEX statement

Use the RENAME INDEX statement to change the name of an existing index. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
RENAME INDEX [ owner . ] old_index TO new_index
```

Element	Description	Restrictions	Syntax
new_index	New name that you declare here for the index	Name must be unique to the database (or to the session, if *old_index* is on a temporary table)	Identifier on page 1231
old_index	Index name that *new_index* replaces	Must exist, but it cannot be any of the following: • An index on a system catalog table • A Virtual-Index Interface (VII)	Identifier on page 1231
owner	Owner of index	Must be the owner of *old_index*	Owner name on page 1262

Usage

You can rename an index if you are the owner of the index or have the DBA privilege on the database.

When you rename an index, the database server changes the index name in the **sysindexes**, **sysconstraints**, **sysobjstate**, and **sysfragments** system catalog tables. (But for an index on a temporary table, no system catalog tables are updated.)

You can rename a system-generated index that implements a primary-key or foreign-key constraint. When you rename a system-generated index, the index is not automatically dropped if you drop the corresponding constraint. The name of a system-generated index consists of a white space, a number, an underscore, and a number. Omit the white space at the beginning of the system-generated index name in the RENAME INDEX statement. For example, to rename an index that is named " 100_1" to idx1, run the following statement:

```
RENAME INDEX 100_1 TO idx1;
```

You can include the owner name before the system-generated index name, for example:

```
RENAME INDEX jmm.100_1 TO idx1;
```

Indexes on system catalog tables cannot be renamed.

By default, SPL routines that use the renamed index are reoptimized when they are next executed after the index is renamed. When automatic recompilation is disabled, however, SPL routines that use the renamed index are automatically recompiled on their next use if the renamed index is associated with a directly referenced table. If the table is only referenced indirectly, however, execution can fail with error -710. For more information about enabling or disabling automatic recompilation after changes to the schema of a referenced table, see the IFX_AUTO_REPREPARE session environment option. For more information about the AUTO_REPREPARE configuration parameter, see your *HCL® Informix® Administrator's Reference*.

RENAME SECURITY statement

Use the RENAME SECURITY statement to change the name of an existing security object. The object can be a security policy, or a security label, or a security label component.

Syntax

```
RENAME SECURITY { POLICY | LABEL policy. | LABEL COMPONENT } old_nameTOnew_name
```

Element	Description	Restrictions	Syntax
new_name	New name that you declare here for the security object	Must be unique among identifiers of security objects in the database, and must be different from *old_name*	Identifier on page 1231
old_name	Current® name that *new_name* replaces	Must exist in the database as the identifiers of a security object	Identifier on page 1231
policy	Security policy of the *old_name* label	Must be the security policy for which *old_name* is a security label	Identifier on page 1231

Usage

This statement is an extension to the ANSI/ISO standard for SQL.

Only DBSECADM can issue this statement. The RENAME SECURITY statement replaces the *old_name* with the specified *new_name* in the table of the system catalog in which the renamed security object is registered:

- **sysecpolicies.secpolicyname** for security policies
- **sysseclabels.seclabelname** for security labels
- **sysseclabelcomponents.compname** for security label components.

This statement does not, however, change the numeric value of the **sysecpolicies.secpolicyid**, **sysseclabels.seclabelid**, nor **sysseclabelcomponents.compid** of the renamed security object.

The keyword or keywords that follow the SECURITY keyword identify the type of security object that is being renamed. In the following example, the new identifier **honesty** replaces **best** as the name of a security policy:

```
RENAME SECURITY POLICY best TO honesty;
```

In the following example, the new identifier **transparent** replaces **opaque** as the name of a label for the **honesty** security policy:

```
RENAME SECURITY LABEL honesty.opaque TO transparent;
```

In the next example, the new identifier **accountant** replaces **architect** as the name of a security label component:

```
RENAME SECURITY LABEL COMPONENT architect TO accountant;
```

RENAME SEQUENCE statement

Use the RENAME SEQUENCE statement to change the name of a sequence. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
RENAME SEQUENCE [ owner . ] old_sequenceTOnew_sequence
```

Element	Description	Restrictions	Syntax
new_sequence	New name that you declare here for an existing sequence	Must be unique among the names of sequences, tables, views, and synonyms in the database	Identifier on page 1231
old_sequence	Current® name of a sequence	Must exist in the current database	Identifier on page 1231
owner	Owner of the sequence	Must be the owner of the sequence	Owner name on page 1262

Usage

To rename a sequence, you must be the owner of the sequence, have the ALTER privilege on the sequence, or have the DBA privilege on the database.

You cannot use a synonym to specify the name of the sequence.

In a database that is not ANSI compliant, the name of *new_sequence* (or in an ANSI-compliant database, the combination of *owner.new_sequence*) must be unique among sequences, tables, views, and synonyms in the database.

RENAME TABLE statement

Use the RENAME TABLE statement to change the name of a table. The RENAME TABLE statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
RENAME TABLE [ owner . ] old_tableTO [ ] new_table
```

Element	Description	Restrictions	Syntax
new_table	New name for *old_table*	Must be unique among the names of sequences, tables, views, and synonyms in the database	Identifier on page 1231
old_table	Name that *new_table* replaces	Must be the name (not the synonym) of a table that is registered in the current database	Identifier on page 1231
owner	Current® owner of the table	Must be the owner of the table.	Owner name on page 1262

Usage

To rename a table, you must be the owner of the table, or have the ALTER privilege on the table, or have the DBA privilege on the database.

An error occurs if *old_table* is a synonym, rather than the name of a table.

The *old_table* can be an object that the CREATE EXTERNAL TABLE statement defined.

The renamed table remains in the current database. You cannot use the RENAME TABLE statement to move a table from the current database to another database, nor to rename a table that resides in another database.

You cannot change the table *owner* by renaming the table. An error occurs if you try to specify an *owner*. qualifier for the new name of the table.

When the table owner is changed, you must specify both the old owner and new owner.

In an ANSI-compliant database, if you are not the owner of *old_table*, you must specify *owner*.*old_table* as the old name of the table.

If *old_table* is referenced by a view in the current database, the view definition is updated in the **sysviews** system catalog table to reflect the new table name. For further information on the **sysviews** system catalog table, see the *HCL® Informix® Guide to SQL: Reference*.

If *old_table* is a triggering table, the database server takes these actions:

- Replaces the name of the table in the trigger definition but does not replace the table name where it appears inside any triggered actions
- Returns an error if the new table name is the same as a correlation name in the REFERENCING clause of the trigger definition

When the trigger executes, the database server returns an error if it encounters a table name for which no table exists.

Using RENAME TABLE to Reorganize a Table

The RENAME TABLE statement can be a useful alternative to the ALTER TABLE statement when you need to reorganize the schema of an existing table. Suppose, for example, that you decide to change the order of columns in the **items** table of the **stores** demonstration database. You can reorganize the **items** table to move the **quantity** column from the fifth position to the third position by following these steps:

1. Create a new table, **new_table**, that contains the column **quantity** in the third position.
2. Fill the table with data from the current **items** table.
3. Drop the old **items** table.
4. Rename **new_table** with the identifier **items**.

The following example uses the RENAME TABLE statement as the last step:

```
CREATE TABLE new_table
   (
   item_num         SMALLINT,
   order_num        INTEGER,
   quantity         SMALLINT,
   stock_num        SMALLINT,
   manu_code        CHAR(3),
   total_price      MONEY(8)
   );
INSERT INTO new_table
   SELECT item_num, order_num, quantity, stock_num,
        manu_code, total_price FROM items;
DROP TABLE items;
RENAME TABLE new_table TO items;
```

RENAME TRUSTED CONTEXT statement

Use the RENAME TRUSTED CONTEXT statement to change the name of a trusted-context object.

This statement is an extension to the ANSI/ISO standard for SQL. You must hold the database security administrator (DBSECADM) role to rename a trusted context.

Syntax

RENAME TRUSTED CONTEXT *old_name* **TO** *new_name*

Element	Description	Restrictions	Syntax
old_name	Trusted context identifier that replaces the data	Must be an existing trusted-context object	Identifier on page 1231
new_name			

Element	Description	Restrictions	Syntax
		ab ase ser ver	
new_name	New name that you declare here for the trusted context	Must be a one-part name, without qualifiers. It cannot begin with the characters "SYS" and must	Identifier on page 1231

Element	Description	Restrictions	Syntax
		not identify a trusted context that already exists on the database server.	

Usage

The *new_name* and the *old_name* cannot include qualifiers, such as *owner*, *database*, or *dbserver*.

After the RENAME TRUSTED CONTEXT statement successfully executes, all references to the *old_name* will be replaced by *new_name* in these tables in the **sysuser** database of the Informix® database server instance:

- **systrustedcontext**
- **systcxattributes**
- **systcxusers**.

In addition, applications that attempt to establish a connection to the database by referencing the *old_name* will fail, unless the *old_name* has been declared as the identifier of a new trusted-context object.

If you rename the trusted context while trusted connections for this context are active, those connections remain trusted until they terminate, or until the next reuse attempt. If an attempt is made to switch the user on these trusted connections, however, an error is returned.

Example

The following example of a complete RENAME TRUSTED CONTEXT statement replaces the security-object identifier cntx1 with cntx2 as the new name for the cntx1 trusted context:

```
RENAME TRUSTED CONTEXT cntx1 TO cntx2;
```

This example fails under either of the following circumstances:

- if cntx1 is not the name of a trusted-context object of the current database server instance,
- or if cntx2 is already the name of an existing trusted-context object of the same database server.

RENAME USER statement (UNIX™, Linux™)

Use the RENAME USER statement to change the name of an internal user of a non-root installation of the database server.

This statement is an extension to the ANSI/ISO standard for the SQL language.

Syntax

RENAME USER *old_name* **TO** *new_name*

Element	Description	Restrictions	Syntax
old_name	Authorization identifier of a specific user that you are renaming.	Must be an existing authorization identifier	Owner name on page 1262
new_name	Authorization identifier of a specific user.	Cannot be an existing authorization identifier	Owner name on page 1262

Usage

Only a DBSA can run the RENAME USER statement. With a non-root installation, the user who installs the server is the equivalent of the DBSA, unless the user delegates DBSA privileges to a different user.

Do not rename a user while the user is active on a connection. Running the statement does not transfer any database or table level privileges granted to the old user name to the new user name.

Execution of the RENAME USER statement can be audited with the RNUR audit code.

The USERMAPPING configuration parameter must be set to BASIC or ADMIN.

You must also enter values in the SYSUSERMAP table of the **sysusers** database to map users with the appropriate user properties so that the mapped user statements of SQL to work correctly.

Example

Example

The following statement renames the user **bill** to **bob**:

```
RENAME USER bill TO bob;
```

REVOKE statement

Use the REVOKE statement to cancel access privileges or roles that are held by users, by roles, or by PUBLIC, or to cancel user security labels or exemptions from the rules of security policies.

Syntax

(explicit id unique_928_Connect_42_revo001b) unique_928_Connect_42_revo001b(explicit id unique_928_Connect_42_revo004b) unique_928_Connect_42_revo004b(explicit id unique_928_Connect_42_revo005b) unique_928_Connect_42_revo005b(explicit id unique_928_Connect_42_revo006b) unique_928_Connect_42_revo006b(explicit id unique_928_Connect_42_revo007b) unique_928_Connect_42_revo007b(explicit id unique_928_Connect_42_revo008b) unique_928_Connect_42_revo008b(explicit id unique_928_Connect_42_revo009b) unique_928_Connect_42_revo009b(explicit id unique_928_Connect_42_revo010b) unique_928_Connect_42_revo010b(explicit id unique_928_Connect_42_revo011b) unique_928_Connect_42_revo011b(explicit id unique_928_Connect_42_revo012b) unique_928_Connect_42_revo012b

```
REVOKE { { [ | { <Database-Level Privileges> [ FROM <User List> [
| { DEFAULT ROLE | ACCESS } FROM { PUBLIC | 'user' } } | { <Role Name> [ FROM { PUBLIC | { | 'user' | 'role' } } [ AS { revoker |
'revoker' } ] | <Security Administration Options> [ } } | { <Table-Level Privileges> [ | { [ | <Routine-Level Privileges> [ | { {
 <Language-Level Privileges> [ | <Type-Level Privileges> [ | <Sequence-Level Privileges> [ } } } } <FROM Options> }
```

FROM options

"FROM"

"{ { <User List> [{ CASCADE | [RESTRICT] }] | [{ | { <Role Name> [| 'user' } } } }] [AS { revoker | 'revoker' }]"

Element	Description	Restrictions	Syntax
revoker	Authorization identifier of the grantor of the privileges to be revoked	Must be grantor of the specified privileges	Owner name on page 1262
role	Role from which you revoke another role	Must exist	Owner name on page 1262
user	User whose role (or default role) you cancel	Must exist	Owner name on page 1262

Usage

To cancel privileges on one or more fragments of a table that has been fragmented by expression, see REVOKE FRAGMENT statement on page 679.

You can revoke privileges if any of the following conditions is true for the privileges that you are attempting to revoke on some database object:

- You granted them and did not designate another user as grantor.
- The GRANT statement specified you as grantor.
- You are revoking privileges from PUBLIC on an object that you own, and those privileges were granted by default when you created the object.
- You have database-level DBA privileges and you specify in the AS clause the name of a user who was grantor of the privilege.

The REVOKE statement can cancel any of the following access privileges or roles that a user, or PUBLIC, or a role currently holds:

- Privileges on the database (but a role cannot hold database-level privileges)
- Privileges on a table, synonym, view, or sequence object
- Privileges on a user-defined data type (UDT), a user-defined routine (UDR), or on the SPL language
- A non-default role, or the default role of PUBLIC or of a user.

You cannot revoke privileges from yourself. You cannot revoke grantor status from another user. To revoke a privilege that was granted to another user by the AS *grantor* clause of the GRANT statement, you must have the DBA privilege, and you must use the AS clause to specify that user as *revoker*.

Delimiting *revoker*, *role*, and *user* identifiers

The REVOKE statement syntax diagram is simplified, because

- the quotation-mark delimiters that can enclose the *revoker*, *role*, or *user* names are optional,
- and you can also substitute double (`"`) quotation marks for single (`'`) quotation marks to delimit those identifiers. Both delimiters must be the same character.

If you enclose *revoker*, *role*, or *user* in quotation marks, the name is case sensitive, and the database server stores it in the system catalog exactly as you enter it in the REVOKE statement. For example, the following statement revokes the DBA privilege from user **`sam`**.

```
REVOKE DBA FROM "sam";
```

If another user who holds the DBA privilege has the authorization identifier **`Sam`**, they would not be affected by the REVOKE example, because their case sensitive user name does not match the delimited name. In a database that was not created as MODE ANSI, however, the following example revokes the DBA privilege of both users **`sam`** and **`Sam`**:

```
REVOKE DBA FROM sam;
```

In an ANSI-compliant database, if you do not use quotation marks as delimiters, that identifier is stored in uppercase letters.

In locales that support letter case, authorization identifiers for distinct users or roles that differ only in letter case might produce unexpected results, if GRANT and REVOKE statements use both delimited and undelimited *user* and *role* specifications to manage access privileges.

Revoking database server access from mapped users (UNIX™, Linux™)

Use the REVOKE ACCESS FROM statement to remove surrogate user properties from specific mapped users.

Only user **informix** or a DBSA can run the REVOKE ACCESS FROM statement.

The REVOKE ACCESS FROM statement does not affect any access privileges of the Informix® user account name that accesses the database server through an OS-level account on the host computer.

Example:

User **informix** or a DBSA can run the following statement on a system that supports mapped users, and one of the mapped users is user **bob**:

```
REVOKE ACCESS FROM bob;
```

This statement entirely removes the access of user **bob** to the database server, except when one or both of the following is true:

- PUBLIC is mapped to surrogate user properties. In this case, user **bob** still retains the same access privileges that the PUBLIC group holds.
- User **bob** is also a user account on the Informix® host computer with database server access.

Database-level privileges

Three concentric layers of database-level privileges, Connect, Resource, and DBA, authorize increasing power over database access and control. Only a user with the DBA privilege can grant or revoke database-level privileges.

Database-Level Privileges

"{ DBA | RESOURCE | CONNECT }"

Because of the hierarchical organization of the privileges (as outlined in the privilege definitions that are described later in this section), if you revoke either the Resource or the Connect privilege from a user with the DBA privilege, the statement has no effect. If you revoke the DBA privilege from a user who has the DBA privilege, the user retains the Connect privilege on the database. To deny database access to a user with the DBA or Resource privilege, you must first revoke the DBA or the Resource privilege and then revoke the Connect privilege in a separate REVOKE statement.

Similarly, if you revoke the Connect privilege from a user who has the Resource privilege, the statement has no effect. If you revoke the Resource privilege from a user, the user retains the Connect privilege on the database.

Only users or PUBLIC can hold database-level privileges. You cannot revoke these privileges from a role, because a role cannot hold database level privileges.

The following table lists the keyword for each database-level privilege.

Privilege	Effect
DBA	Has all the capabilities of the Resource privilege and can perform the following additional operations: • Grant any database-level privilege, including the DBA privilege, to another user. • Grant any table-level privilege to another user or to a role. • Grant a role to a user or to another role. • Revoke a privilege whose grantor you specify as the *revoker* in the AS clause of the REVOKE statement. • Restrict the Execute privilege to DBAs when registering a UDR. • Execute the SET SESSION AUTHORIZATION statement. • Create any database object. • Create tables, views, and indexes, designating another user as owner of these objects. • Alter, drop, or rename database objects, regardless of who owns it. • Execute the DROP DISTRIBUTIONS option of the UPDATE STATISTICS statement. • Execute DROP DATABASE and RENAME DATABASE statements.
RESOURCE	Lets you extend the structure of the database. In addition to the capabilities of the Connect privilege, the holder of the Resource privilege can perform the following operations: • Create new tables. • Create new indexes. • Create new user-defined routines. • Create new data types.
CONNECT	If you have this privilege, you can query and modify data, and modify the database schema if you own the database object that you want to modify. A user holding the Connect privilege can perform the following operations: • Connect to the database with the CONNECT statement or another connection statement. • Execute SELECT, INSERT, UPDATE, and DELETE statements, provided that the user has the necessary table-level privileges. • Create views, provided that the user has the Select privilege on the underlying tables. • Create synonyms. • Create temporary tables and create indexes on temporary tables. • Alter or drop a table or an index, if the user owns the table or index (or has the Alter, Index, or References privilege on the table). • Grant privileges on a table, if the user owns the table (or was given privileges on the table with the WITH GRANT OPTION keyword).

Tip: To determine which users have DBA privileges on a database, run this query from DB-Access or your application:

```
select username,usertype from sysusers;
```

The output shows user names (for example, public and informix) followed by one of the following codes:

- D = DBA privilege
- C = Connect privilege
- R = Resource privilege

Table-Level Privileges

Table-level privileges, also called *table privileges*, specify which operations a user or role can perform on a table or view in the database. You can use a synonym to specify the table or view on which you grant or revoke table privileges.

Select, Update, and References privileges can be granted on a subset of the columns of a table or view, but can be revoked only for all columns. If Select privileges are revoked from a user for a table that is referenced in the SELECT statement defining a view that the same user owns, then that view is dropped, unless it also includes columns from tables in another database.

For table objects that the CREATE EXTERNAL TABLE statement has registered in the current database, only the Select privilege and the Insert privilege are supported; no other table or column access privileges can be granted or revoked.

Use the following syntax to specifying which table-level privileges to revoke from one or more users or roles:

(explicit id unique_931_Connect_42_tlp003b) unique_931_Connect_42_tlp003b

Table-Level Privileges

" { **ALL PRIVILEGES** | { | { **INSERT** | **DELETE** | **UPDATE** | { [] | **SELECT** | **ALTER** | **INDEX** | **REFERENCES** | **UNDER** } } } } "

" **ON** "

" [*owner* .] "

" { *table* | *view* | *synonym* } "

Element	Description	Restrictions	Syntax
owner	Name of the user who owns the *table*, *view*, or *synonym*	Must be a valid authorization identifier	Owner name on page 1262
synonym, *table*, *view*	Synonym, table, or view on which privileges are granted	Must exist in the current database	Identifier on page 1231

In one REVOKE statement, you can list one or more of the following keywords to specify the privileges on the specified table to be revoked from the users or roles.

Privilege	Effect after REVOKE
INSERT	User cannot insert rows.
DELETE	User cannot delete rows.
SELECT	User cannot display data retrieved by a SELECT statement.
UPDATE	User cannot change column values.
INDEX	User cannot create permanent indexes. You must have the Resource privilege to take advantage of the Index privilege. (But any user who has the Connect privilege can create indexes on temporary tables.)
ALTER	The holder cannot add or delete columns, modify column data types, add or delete constraints, change the locking mode of a table from PAGE to ROW, nor add or drop a corresponding named ROW type table. The user also cannot enable or disable indexes, constraints, nor triggers, as described in SET Database Object Mode statement on page 784.

Privilege	Effect after REVOKE
REFERENCES	User cannot reference columns in referential constraints. You must also have the Resource privilege on the database to take advantage of the References privilege on tables. (You can add, however, a referential constraint during an ALTER TABLE statement. without holding the Resource privilege on the database.) Revoking the References privilege disallows cascading DELETE operations.
UNDER	User cannot create subtables under a typed table.
ALL	This removes all of the table privileges that are listed above. (Here the PRIVILEGES keyword is optional.)

See also Table-Level Privileges on page 540.

If a user receives the same privilege from two different grantors and one grantor revokes the privilege, the grantee still has the privilege until the second grantor also revokes the privilege. For example, if both you and a DBA grant the Update privilege on your table to **ted**, both you and the DBA must revoke the Update privilege to prevent **ted** from updating your table.

If user **ted** holds the same privileges through a role or as PUBLIC, however, this REVOKE operation does not prevent **ted** from exercising the Update privilege.

When to Use REVOKE Before GRANT

You can use combinations of REVOKE and GRANT to replace PUBLIC with specific users as grantees, and to remove table-level privileges on some columns.

Replacing PUBLIC with Specified Users

If a table owner grants a privilege to PUBLIC, the owner cannot revoke the same privilege from any specific user. For example, assume PUBLIC has default Select privileges on your **customer** table. Suppose that you issue the following statement in an attempt to exclude **ted** from accessing your table:

```
REVOKE ALL ON customer FROM ted;
```

This statement results in error message 111, `No record found`, because the system catalog tables (**syscolauth** or **systabauth**) contain no table-level privilege entry for a user named **ted**. This REVOKE operation does not prevent **ted** from keeping all the table-level privileges given to PUBLIC on the **customer** table.

To restrict table-level privileges, first revoke the privileges with the PUBLIC keyword, then re-grant them to some appropriate list of users and roles. The following statements revoke the Index and Alter privileges from all users for the **customer** table, and then grant these privileges specifically to user **mary**:

```
REVOKE INDEX, ALTER ON customer FROM PUBLIC;
GRANT INDEX, ALTER ON customer TO mary;
```

Restricting Access to Specific Columns

Unlike GRANT, the REVOKE statement has no syntax to specify privileges on a subset of columns in a table. To revoke the Select, Update, or References privilege on a column from a user, you must revoke the privilege for all the columns of the table. To provide access to some of the columns on which you previously had granted privileges, issue a new GRANT statement to restore the appropriate privilege on specific columns.

The next example cancels Select privileges for PUBLIC on certain columns:

```
REVOKE SELECT ON customer FROM PUBLIC;
GRANT SELECT (fname, lname, company, city) ON customer TO PUBLIC;
```

In the next example, **mary** first receives the ability to reference four columns in **customer**, then the table owner restricts references to two columns:

```
GRANT REFERENCES (fname, lname, company, city) ON customer TO mary;
REVOKE REFERENCES ON customer FROM mary;
GRANT REFERENCES (company, city) ON customer TO mary;
```

Effect of the ALL Keyword

The ALL keyword revokes all table-level privileges. If any or all of the table-level privileges do not exist for the revokee, REVOKE with the ALL keyword executes successfully but returns the following **SQLSTATE** code:

```
01006--Privilege not revoked
```

For example, assume that user **hal** has the Select and Insert privileges on the **customer** table. User **jocelyn** wants to revoke all table-level privileges from user **hal**. So user **jocelyn** issues the following REVOKE statement:

```
REVOKE ALL ON customer FROM hal;
```

This statement executes successfully but returns **SQLSTATE** code 01006. The **SQLSTATE** warning is returned because both of the following are true:

- The statement succeeds in revoking the Select and Insert privileges from user **hal** because user **hal** had those privileges.
- **SQLSTATE** code 01006 is returned because user **hal** lacked other privileges implied by the ALL keyword, but these privileges were not revoked.

The ALL keyword instructs the database server to revoke everything possible, including nothing. If the user from whom privileges are revoked has no privileges on the table, the REVOKE ALL statement still succeeds, because it revokes everything possible from the user (in this case, no privileges at all).

Effect of the ALL Keyword on UNDER Privilege

If you revoke ALL privileges on a typed table, the Under privilege is included in the privileges that are revoked. If you revoke ALL privileges on a table that is not based on a ROW type, the Under privilege is not included among the privileges that are revoked. (The Under privilege can be granted only on a typed table.)

Effect of Uncommitted Transactions

The REVOKE statement places an exclusive row lock on the entry in the **systables** system catalog table for the table on which privileges are revoked. This lock is not released until the transaction that contains the REVOKE statement terminates. When another transaction attempts to prepare a SELECT statement against this table while the first transaction is open, the concurrent transaction fails, because the **systables** row for the specified table remains exclusively locked. The attempt to prepare the SELECT statement cannot succeed until after the first transaction is either committed or rolled back.

Type-Level Privileges

You can revoke two privileges on data types:

- The Usage privilege on a user-defined data type
- The Under privilege on a named ROW type

Type-Level Privileges

" { **USAGE ON TYPE***type_name* | **UNDER ON TYPE***row_type* } "

Element	Description	Restrictions	Syntax
row_type	Named ROW type for which to revoke Under privilege	Must exist	Data Type, p. Data Type on page 985
type_name	User-defined type for which to revoke Usage privilege	Must exist	Data Type, p. Data Type on page 985

Usage Privilege

Any user can reference a built-in data type in an SQL statement, but not a DISTINCT data type that is based on a built-in data type. The creator of a user-defined data type or a DBA must explicitly grant the Usage privilege on the UDT, including a DISTINCT data type based on a built-in data type.

REVOKE with the USAGE ON TYPE keywords removes the Usage privilege that you granted earlier to another user, to PUBLIC, or to a role.

The following statement revokes from user **mark** the privilege of using the **widget** user-defined type:

```
REVOKE USAGE ON TYPE widget FROM mark;
```

Under Privilege

You own any named ROW data type that you create. If you want other users to be able to create subtypes under this named ROW type, you must grant these users the Under privilege on your named ROW type. If you later want to remove the ability of these users to create subtypes under the named ROW type, you must revoke the Under privilege from these users. A REVOKE statement with the UNDER ON TYPE keywords removes the Under privilege that you granted earlier to these users.

For example, suppose that you created a ROW type named **rtype1**:

```
CREATE ROW TYPE rtype1 (cola INT, colb INT);
```

If you want another user named **kathy** to be able to create a subtype under this named ROW type, you must grant the Under privilege on this named ROW type to user **kathy**:

```
GRANT UNDER ON TYPE rtype1 TO kathy;
```

Now user **kathy** can create another ROW type under the **rtype1** ROW type even though **kathy** is not the owner of the **rtype1** ROW type:

```
CREATE ROW TYPE rtype2 (colc INT, cold INT) UNDER rtype1;
```

If you later want to remove the ability of user **kathy** to create subtypes under the **rtype1** ROW type, enter the following statement:

```
REVOKE UNDER ON TYPE rtype1 FROM kathy;
```

Routine-Level Privileges

If you revoke the Execute privilege on a UDR from a user, that user can no longer execute that UDR in any way. For details of how a user can execute a UDR, see Routine-Level Privileges on page 546.

(explicit id unique_933_Connect_42_rolp002) unique_933_Connect_42_rolp002(explicit id unique_933_Connect_42_rolp003) unique_933_Connect_42_rolp003

Routine-Level Privileges

" **EXECUTE ON** "

" { *SPL_routine* | { **PROCEDURE** | **FUNCTION** | **ROUTINE** } *routine* ([<Routine Parameter List>]) | **SPECIFIC** { **ROUTINE** | **FUNCTION** | **PROCEDURE** } <Specific Name> } "

Element	Description	Restrictions	Syntax
routine	A user-defined routine	Must exist	Identifier on page 1231
SPL_routine	An SPL routine	Must be unique in the database	Identifier on page 1231

In an ANSI-compliant database, the *owner* name must qualify the *routine* name, unless the user who issues the REVOKE statement is the owner of the routine.

The following example cancels the Execute privilege of user **mark** on the **delete_order** routine that is owned by **luke**:

```
REVOKE EXECUTE ON ROUTINE luke.delete_order FROM mark;
```

In Informix®, any negator function for which you grant the Execute privilege requires a separate, explicit, REVOKE statement.

When you create a UDR under any of the following circumstances, PUBLIC is not granted Execute privilege by default. Therefore you must explicitly grant the Execute privilege before you can revoke it:

- You create the UDR in an ANSI-compliant database.
- You have DBA privilege and specify DBA after the CREATE keyword to restrict the Execute privilege to users with the DBA database-level privilege.
- The **NODEFDAC** environment variable is set to `yes` to prevent the PUBLIC group from receiving any access privileges by default that are not explicitly granted.

Setting **NODEFDAC** to `yes` also prevents PUBLIC from receiving table access privileges by default when a new table is created in a database that was not created as mode ANSI. The **NODEFDAC** setting, however, cannot prevent the PUBLIC group from being granted the same privileges by a user who holds the necessary access privileges on the new UDR or on the new table.

If you create a UDR with none of the above conditions in effect, however, PUBLIC can execute your UDR without the GRANT EXECUTE statement. To limit who can execute your UDR, revoke Execute privilege FROM PUBLIC, and grant it to users (see User List on page 666) or roles (see Role Name on page 667).

In Informix®, if two or more UDRs have the same name, use a keyword from this list to specify which of those UDRs a user list can no longer execute.

Keyword

UDR for Which Execution by the User is Prevented

SPECIFIC

The UDR identified by *specific name*

FUNCTION

Any function with the specified *routine name* (and parameter types that match *routine parameter list*, if specified)

PROCEDURE

Any procedure with the specified *routine name* (and parameter types that match *routine parameter list*, if specified)

ROUTINE

Functions or procedures with the specified *routine name* (and parameter types that match *routine parameter list*, if specified)

Language-Level Privileges

To register or drop a UDR written in the SPL, C, or Java™ languages, a user must hold the Usage privilege on the programming language in which the routine is written.

This is the syntax of the USAGE ON LANGUAGE clause for specifying a language-level privilege to revoke:

Language-Level Privileges

"USAGE ON LANGUAGE"

"{SPL | C | JAVA}"

Each REVOKE USAGE ON LANGUAGE statement can specify no more than one programming language.

When a user registers a UDR that is written in the SPL, C, or Java™ language, the database server verifies that the user has the Usage privilege on the language in which the UDR is written. If the user does not, the CREATE FUNCTION or CREATE PROCEDURE statement fails with an error. If the IFX_EXTEND_ROLE configuration parameter has enabled the built-in EXTEND role, only users who also hold that role can register or drop UDRs written in the C language or in the Java™ language, even if the users hold USAGE ON LANGUAGE privileges for those languages.

To revoke the Usage privilege on a programming language from a user or role, issue a REVOKE statement that includes the USAGE ON LANGUAGE keywords and a keyword that specifies the programming language. If this statement succeeds, any user or role that you specify in the FROM clause can no longer register UDRs that are written in the specified language. For example, if you revoke the default Usage privilege on SPL from PUBLIC, the ability to create SPL routines is taken away from all users (except those who have been individually granted the Usage privilege on the SPL language, or who hold that Usage privilege through a role:

```
REVOKE USAGE ON LANGUAGE SPL FROM PUBLIC;
```

You can issue a GRANT USAGE ON LANGUAGE statement to restore Usage privilege on SPL to a restricted group, such as to the role named **developers**:

```
GRANT USAGE ON LANGUAGE SPL TO developers;
```

Sequence-Level Privileges

Although Informix® implements sequence objects as tables, only the following subset of the table privileges (as described in Table-Level Privileges on page 540) can be granted or revoked on a sequence:

- Select privilege
- Alter privilege

Use the following syntax to specify privileges to revoke on a sequence object:

Sequence-Level Privileges

"{ ALL | { | ALTER | SELECT } }"

"ON"

"[*'owner.'*]"

"{ *sequence* | *synonym* }"

Element	Description	Restrictions	Syntax
owner	Owner of the sequence or of its synonym	Must be the owner	Owner name on page 1262
sequence	Sequence on which to revoke privileges	Must exist	Identifier on page 1231
synonym	Synonym for a sequence object	Must point to a sequence	Identifier on page 1231

The sequence must reside in the current database. (You can qualify the *sequence* or *synonym* identifier with a valid *owner* name, but the name of a remote *database* (or *database*@*server*) is not valid as a qualifier.) Syntax to revoke sequence-level privileges is an extension to the ANSI/ISO standard for SQL.

Alter Privilege

You can revoke the Alter privilege on a sequence from another user, from PUBLIC, or from a role. The Alter privilege enables a specified user or role to modify the definition of a sequence with the ALTER SEQUENCE statement or to rename the sequence with the RENAME SEQUENCE statement.

The following REVOKE statement cancels any Alter privilege that was granted individually to user **mark** on the **cust_seq** sequence object:

```
REVOKE ALTER ON cust_seq FROM mark;
```

Select Privilege

You can revoke the Select privilege on a sequence from another user, from PUBLIC, or from a role. Select privilege enables a user or role to use the *sequence*.**CURRVAL** and *sequence*.**NEXTVAL** in SQL statements to access and to increment the value of a sequence.

The following REVOKE statement cancels any Select privilege that was granted individually to user **mark** on the **cust_seq** sequence object:

```
REVOKE SELECT ON cust_seq FROM mark;
```

ALL Keyword

You can use the ALL keyword to revoke both Alter and Select privileges from another user, from PUBLIC, or from a role.

The following example cancels any Alter and Select privileges that user **mark** holds on the **cust_seq** sequence object:

```
REVOKE ALL ON cust_seq FROM mark;
```

Whether **mark** can still access **cust_seq** after this statement executes depends on whether the user still holds Alter or Select privileges on **cust_seq** that were granted to PUBLIC, or if he holds a role to which unrevoked privileges on **cust_seq** have been granted.

User List

The authorization identifiers (or the PUBLIC keyword) that follow the FROM keyword of REVOKE specify who loses the revoked privileges or revoked roles. If you use the PUBLIC keyword as the user list, the REVOKE statement revokes the specified privileges or roles from PUBLIC, thereby revoking them from all users to whom the privileges or roles have not been explicitly granted, or who do not hold some other role through which they have received the role or privilege.

The *user list* can consist of the authorization identifier of a single user or of multiple users, separated by commas. If you use the PUBLIC keyword as the user list, the REVOKE statement revokes the specified privileges from all users.

User List

" { **PUBLIC** | { | *user* | *'user'* } } "

Element	Description	Restrictions	Syntax
user	Login name of a user whose privilege or role you are revoking	Must be a valid authorization identifier	Owner name on page 1262

Spell the user names in the list exactly as they were spelled in the GRANT statement. You can optionally use quotation marks around each user name in the list to preserve the lettercase. In an ANSI-compliant database, if you do not use quotation marks to delimit *user*, the name of the user is stored in uppercase letters unless the **ANSIOWNER** environment variable was set to 1 before the database server was initialized.

When you specify login names, you can use the REVOKE statement and the GRANT statement to secure various types of database objects selectively. For examples, see When to Use REVOKE Before GRANT on page 659.

Role Name

Only the DBA or a user who was granted a role WITH GRANT OPTION can revoke a role or its privileges. Users cannot revoke roles from themselves.

Role Name

" { *'role'* | *role* } "

Element	Description	Restrictions	Syntax
role	A role with one of these attributes: • Loses an existing privilege or role • Is lost by a user or by another role	Must exist. If enclosed between quotation marks, *role* is case sensitive.	Owner name on page 1262

Immediately after the REVOKE keyword, the name of a *role* specifies a role to be revoked from the user list. After the FROM keyword, however, the name of a *role* specifies a role from which access privilege (or another role) is to be revoked. The same FROM clause can include both *user* and *role* names if no other REVOKE options conflict with the *user* or *role* specifications. Syntax to revoke privileges on a role or from a role are extensions to the ANSI/ISO standard for SQL.

When you include a *role* after the FROM keyword of the REVOKE statement, the specified privilege (or another role) is revoked from that role, but users who have that role retain any privileges or roles that were granted to them individually.

If you enclose *role* between quotation marks, the name is case sensitive and is stored exactly as you typed it. In an ANSI-compliant database, if you do not use quotation marks as delimiters, the *role* is stored in uppercase letters.

When you revoke a role that was granted to a user with the WITH GRANT OPTION keywords, you revoke both the role and the option to grant it.

The following examples show the effects of REVOKE *role*:

- Remove users or remove another role from inclusion in the specified role:

```
REVOKE accounting FROM mary;
REVOKE payroll FROM accounting;
```

- Remove one or more access privileges from a role:

```
REVOKE UPDATE ON employee FROM accounting;
```

When you revoke table-level privileges from a role, you cannot include the RESTRICT or CASCADE keywords.

Revoking a Default Role

The DBA or the owner of the database can define a *default role* for one or more users or for PUBLIC with the GRANT DEFAULT ROLE statement. Unlike a non-default role, which does not take effect until the SET ROLE statement activates the role, a default role takes effect automatically when the user connects to the database. The default role can specify a set of access privileges for all the users who are granted that default role. Conversely, the REVOKE DEFAULT ROLE statement cancels the current default role as the default role for the specified *user-list*, as in the following program fragment:

```
CREATE ROLE accounting;
GRANT USAGE ON LANGUAGE SPL TO accounting;
GRANT ALL PRIVILEGES ON receivables TO accounting;
GRANT DEFAULT ROLE accounting TO mary;
. . .
REVOKE DEFAULT ROLE FROM mary;
```

The last statement removes from user **mary** any access privileges that they hold only through her default role. In this example, the default role was **accounting**, but because at a given point in time there can be only one default role for an individual user (or for the PUBLIC group), the name of the default role is not specified in the REVOKE DEFAULT ROLE statement. If **mary** issues the SET ROLE DEFAULT statement, it has no effect until they are granted some new default role.

After you execute REVOKE DEFAULT ROLE specifying one or more users or PUBLIC, any privileges that those users held only through the default role are cancelled. (But this statement does not revoke any privileges that were granted to a user individually, or privileges that were granted to a user through another role, or privileges that PUBLIC holds.)

After REVOKE DEFAULT ROLE successfully cancels the default role of *user*, the default role of *user* becomes NULL, and the default role information is removed from the system catalog. (In this context, NULL and NONE are synonyms.)

No warning is issued if REVOKE DEFAULT ROLE specifies a user who has not been granted a default role.

No options besides the *user-list* are valid after the FROM keyword in the REVOKE DEFAULT ROLE statement.

Revoking the EXTEND Role

The REVOKE EXTEND FROM *user-list* statement cancels the EXTEND role of the specified users. In databases where the IFX_EXTEND_ROLE configuration parameter is enabled, revoking this role prevents the specified users from creating or dropping external UDRs. Whether or not a user holds the EXTEND role has no effect on creating or dropping UDRs written in the SPL language.

Only the Database Server Administrator (DBSA), by default user **informix**, can grant the built-in EXTEND role to one or more users or to PUBLIC by issuing the GRANT EXTEND TO *user-list* statement. (Because EXTEND is a built-in role, a user who holds it does not need to activate it with the SET ROLE statement, and the DROP ROLE statement cannot destroy the EXTEND role.)

If the IFX_EXTEND_ROLE configuration parameter is set to `ON` or to `1`, users who do not hold the EXTEND role cannot create or drop UDRs that are written in the C or Java™ languages, both of which support shared libraries. The following example revokes the EXTEND role from user **max**:

```
REVOKE EXTEND FROM 'max';
```

This prevents user **max** from creating or dropping external UDRs, even if **max** is the owner of a UDR that he subsequently attempts to drop.

In databases for which this security feature is not needed, the DBSA can disable this restriction on who can create or drop external UDRs by setting the IFX_EXTEND_ROLE parameter to `OFF` or to `0` in the ONCONFIG file. But whether IFX_EXTEND_ROLE is enabled or disabled, users who create or drop external UDRs must also hold the following access privileges:

- Either the Resource privilege or the DBA privilege on the database in which the UDR is registered.
- The Usage privilege on the external programming language in which the UDR is written,

See Database-level privileges on page 656 for information about the Resource privilege. See Language-Level Privileges on page 549 for the syntax of the GRANT USAGE ON LANGUAGE C and the GRANT USAGE ON LANGUAGE JAVA statements of SQL.

Revoking privileges granted WITH GRANT OPTION

If you revoke from *user* privileges or a role that you granted using the WITH GRANT OPTION keywords, you sever the chain of privileges granted by that *user*.

Thus, when you revoke privileges from users or from a role, you also revoke the same privilege resulting from GRANT statements in the following contexts:

- Issued by your grantee
- Allowed because your grantee specified the WITH GRANT OPTION clause
- Allowed because subsequent grantees granted the same privilege or role using the WITH GRANT OPTION clause

The WITH GRANT OPTION clause is only valid in GRANT statements that assign privileges to specific users. The grantee cannot be the PUBLIC group or a role.

The following examples show the revocation of privileges. Suppose you, as the owner of the table **items**, issue the following statements to grant access privileges to user **mary**:

```
REVOKE ALL ON items FROM PUBLIC;
GRANT SELECT, UPDATE ON items TO mary WITH GRANT OPTION;
```

User **mary** then uses their new privilege to grant users **cathy** and **paul** access to the **items** table:

```
GRANT SELECT, UPDATE ON items TO cathy;
GRANT SELECT ON items TO paul;
```

Later you revoke privileges on the **items** table from user **mary**:

```
REVOKE SELECT, UPDATE ON items FROM mary;
```

This single statement effectively revokes all privileges on the **items** table from users **mary**, **cathy**, and **paul**.

The CASCADE keyword has the same effect as this default condition.

The AS Clause

Without the AS clause, the user who executes the REVOKE statement must be the grantor of the privilege that is being revoked. The DBA or the owner of the object can use the AS clause to specify another user (who must be the grantor of the privilege) as the revoker of the privileges.

For example, if user **falstaff** is the owner of table **CONS_table**, and has granted

The AS clause provides the only mechanism by which discretionary access privileges can be revoked on a database object whose *owner* is an authorization identifier, such as **informix**, that is not also a valid user account known to the operating system.

For contexts where the AS *revoker* clause is required, rather than optional, see Revoking the Execute privilege from PUBLIC on page 549.

Effect of CASCADE Keyword on UNDER Privileges

If you revoke the Under privilege on a typed table with the CASCADE option, the Under privilege is removed from the specified user, and any subtables created under the typed table by that user are dropped from the database.

If you revoke the Under privilege on a named ROW type with the CASCADE option when that data type is in use, the REVOKE fails. This exception to the default behavior of the CASCADE option occurs because the database server supports the DROP ROW TYPE statement with the RESTRICT keyword only.

For example, assume that user **jeff** creates a ROW type named **rtype1** and grants the Under privilege on that ROW type to user **mary**. User **mary** now creates a ROW type named **rtype2** under ROW type **rtype1** and grants the Under privilege on ROW type **rtype2** to user **andy.** Then user **andy** creates a ROW type named **rtype3** under ROW type **rtype2**.

If user **jeff** now tries to revoke the Under privilege on ROW type **rtype1** from user **mary** with the CASCADE option, the REVOKE statement fails, because ROW type **rtype2** is still in use by ROW type **rtype3**.

Controlling the Scope of REVOKE with the RESTRICT Option

The RESTRICT keyword causes the REVOKE statement to fail when any of the following dependencies exist:

- A view depends on a Select privilege that you are attempting to revoke.
- A foreign-key constraint depends on a References privilege that you attempt to revoke.
- You attempt to revoke a privilege from a user who subsequently granted this privilege to another user or to a role.

REVOKE does not fail if it specifies a user who has the right to grant the privilege to others but has not exercised that right. For example, assume that user **clara** specifies WITH GRANT OPTION when they grant the Select privilege on the **customer** table to user **ted**. Further assume that user **ted**, in turn, grants the Select privilege on the **customer** table to user **tania**. The following statement that **clara** issued has no effect, because **ted** has used his authority to grant the Select privilege:

```
REVOKE SELECT ON customer FROM ted RESTRICT;
```

In contrast, if user **ted** does not grant the Select privilege to **tania** or to any other user, the same REVOKE statement succeeds. Even if **ted** does grant the Select privilege to another user, either of the following statements succeeds:

```
REVOKE SELECT ON customer FROM ted CASCADE;
REVOKE SELECT ON customer FROM ted;
```

Security Administration Options

In conjunction with the GRANT statement, the REVOKE statement supports the discretionary access control (DAC) data security feature of Informix® by specifying which users or roles hold privileges that are required to access the database or objects within the database.

The Security Administration Options of the REVOKE statement, like their counterparts for the GRANT statement, support an additional set of data security features, called label-based access control (LBAC). These features enable Informix® to allow or withhold access to protected data on the basis of a comparing a row security label or column security label that is contained in the data object to the user security label and other credentials that have been granted to the user who is seeking access.

(explicit id unique_937_Connect_42_rsao002) unique_937_Connect_42_rsao002(explicit id unique_937_Connect_42_rsao003)

unique_937_Connect_42_rsao003(explicit id unique_937_Connect_42_rsao004) unique_937_Connect_42_rsao004(explicit id unique_937_Connect_42_rsao005)

unique_937_Connect_42_rsao005(explicit id unique_937_Connect_42_rsao006) unique_937_Connect_42_rsao006

Security Administration Options

" { <DBSECADM Clause> [] | <EXEMPTION Clause> [] | <SECURITY LABEL Clause> [] | <SETSESSIONAUTH Clause> [] } "

Use of these REVOKE statement security administration options is restricted:

- Only the Database Server Administrator (DBSA), by default user **informix**, can use the REVOKE DBSECADM statement to revoke the DBSECADM role.
- Only a user who holds the DBSECADM role can issue the REVOKE EXEMPTION, REVOKE SECURITY LABEL, or REVOKE SETSESSIONAUTH statements.

DBSECADM Clause

The REVOKE DBSECADM statement prevents the user to whom the DBSECADM role was granted from issuing DDL statements that can create, alter, rename, or drop security objects, including security policies, security labels, and security components.

DBSECADM Clause

" DBSECADM FROM "

" [USER] *user* "

Element	Description	Restrictions	Syntax
user	User from whom the role is to be revoked	Must be the authorization identifier of a user	Owner name on page 1262

The DBSECADM role is a built-in role that only the DBSA can revoke. Unlike user-defined roles, whose scope is the database in which the role is created, the scope of the DBSECADM role is all of the databases of the Informix® instance. It is not necessary for DBSA to reissue the REVOKE DBSECADM statement in other databases of the same server.

Only a user who holds the DBSECADM role can issue the following SQL statements that create or modify security objects:

- ALTER SECURITY LABEL COMPONENT
- CREATE SECURITY LABEL
- CREATE SECURITY LABEL COMPONENT
- CREATE SECURITY POLICY
- DROP SECURITY LABEL
- DROP SECURITY LABEL COMPONENT
- DROP SECURITY POLICY
- RENAME SECURITY LABEL
- RENAME SECURITY LABEL COMPONENT
- RENAME SECURITY POLICY

Only a user who holds the DBSECADM role can use the following SQL statements to reference tables that are protected by a security policy:

- ALTER TABLE ... ADD SECURITY POLICY
- ALTER TABLE ... ADD ... IDSSECURITYLABEL [DEFAULT *label*]
- ALTER TABLE ... ADD ... [COLUMN] SECURED WITH
- ALTER TABLE ... DROP SECURITY POLICY
- ALTER TABLE ... MODIFY ... [COLUMN] SECURED WITH
- ALTER TABLE ... MODIFY ... DROP COLUMN SECURITY
- CREATE TABLE ... COLUMN SECURED WITH
- CREATE TABLE ... IDSSECURITYLABEL [DEFAULT *label*]
- CREATE TABLE ... SECURITY POLICY

The following GRANT and REVOKE statements also cannot be issued by a user who does not hold the DBSECADM role:

- GRANT EXEMPTION
- GRANT SECURITY LABEL
- GRANT SETSESSIONAUTH
- REVOKE EXEMPTION
- REVOKE SECURITY LABEL
- REVOKE SETSESSIONAUTH

The USER keyword that can follow the FROM keyword is optional, and has no effect, but any authorization identifier that the DBSA specifies in the REVOKE DBSECADM statement must be the identifier of an individual user, rather than the identifier of a role. The *user* cannot be the DBSA who issues this REVOKE DBSECADM statement.

In the following example, the DBSA cancels the DBSECADM role of user **niccolo**:

```
REVOKE DBSECADM FROM niccolo;
```

If this statement executes successfully, user **niccolo** can no longer perform the operations listed above.

After the DBSECADM role is revoked, only the DBSA can grant it again to the user from whom it was revoked.

EXEMPTION Clause

The REVOKE EXEMPTION statement modifies the security credentials of the specified user (or list of users) by enabling one or all of the rules of a specified security policy from which the user had been exempt.

EXEMPTION Clause

" **EXEMPTION ON RULE** "

" { **IDSLBACREADARRAY** | **IDSLBACREADTREE** | **IDSLBACREADSET** | **IDSLBACWRITEARRAY** [{ **WRITEDOWN** | **WRITEUP** }] | **IDSLBACWRITESET** | **IDSLBACWRITETREE** | **ALL** } "

" **FOR***policy* "

" **FROM** [**USER**] *user* "

Element	Description	Restrictions	Syntax
policy	Security policy for which the exemption is revoked	Must exist in the database	Identifier on page 1231
user	User to whom the exemption is to be revoked	Must be the authorization identifier of a user	Owner name on page 1262

Only a user who holds the DBSECADM role can issue the REVOKE EXEMPTION statement.

Rules on Which Exemptions Are Revoked

The keyword that follows the ON keyword specifies the predefined access rule of the security policy (whose identifier follows the FOR keyword) for which an exemption is cancelled. The access rule for which exemption is revoked applies when a table that is protected by the specified policy is accessed by a user from whom the exemption is revoked. For descriptions of the predefined rules for read access and for write access that are associated with a security policy, see the section Rules Associated with a Security Policy on page 301.

The following keywords of the REVOKE EXEMPTION statement identify specific **IDSLBACRULES** rules that this statement can apply to formerly exempt users:

- IDSLBACREADARRAY applies to the user the **IDSLBACREADARRAY** rule for the specified security policy. For a user with no exemption, this rule requires that each array component of the user security label must be greater than or equal to the corresponding array component of the data row security label.
- IDSLBACREADSET applies to the user the **IDSLBACREADSET** rule for the specified security policy. For a user with no exemption, this rule requires that each set component of the user security label must include the set component of the data row security label
- IDSLBACREADTREE applies to the user the **IDSLBACREADTREE** rule for the specified security policy. For a user with no exemption, this rule requires that each tree component of the user security label must include at least one of the elements in the tree component of the data row security label, or else an ancestor of one such element.
- IDSLBACWRITEARRAY WRITEDOWN exempts the user from one aspect of the **IDSLBACWRITEARRAY** rule for the specified security policy. The user who loses this exemption cannot write to a row protected by a label that includes an array component level below the level in the label of the user.
- IDSLBACWRITEARRAY WRITEUP exempts the user from one aspect of the **IDSLBACWRITEARRAY** rule for the specified security policy. The user who loses this exemption cannot write to a row protected by a label that includes an array component level above the level in the label of the user.
- IDSLBACWRITEARRAY (with no WRITEDOWN or WRITEUP keyword) applies to the user the **IDSLBACWRITEARRAY** rule for the specified security policy. The user who loses this exemption cannot write to a row whose array component level is above or below the level in the label of the user. .
- IDSLBACWRITESET applies to the user the **IDSLBACWRITESET** rule for the specified security policy. For a user with no exemption, that rule requires that each set component of the user security label must include the set component of the data row security label
- IDSLBACWRITETREE applies to the user the **IDSLBACWRITETREE** rule for the specified security policy. For a user with no exemption, that rule requires that each tree component of the user security label must include at least one of the elements in the tree component of the data row security label, or the ancestor of one such element.
- ALL revokes an exemption from all **IDSLBACRULES** rules for the specified security policy.

In the following example, DBSECADM revokes an exemption from all of the rules of the **MegaCorp** security policy from users **manoj** and **sam**:

```
REVOKE EXEMPTION ON RULE ALL FOR MegaCorp FROM manoj, sam;
```

Security Policies and Grantees of Exemptions

An exemption applies only to the rules of a single security policy. whose name follows the FOR keyword. Because a protected table can have multiple security labels, but no more than one security policy, revocation of an exemption can prevent a user with insufficient security credentials from accessing data in tables that are protected by the specified security policy.

The REVOKE EXEMPTION statement fails with an error if the specified policy does not exist in the database.

The USER keyword that can follow the FROM keyword is optional, and has no effect, but any authorization identifier specified in the REVOKE EXEMPTION statement must be the identifier of an individual user, rather than the identifier of a role. This *user* cannot be the DBSECADM who issues the same REVOKE EXEMPTION statement.

In the following example, DBSECADM revokes an exemption from user **lynette** for rule **IDSLBACREADARRAY** of the **MegaCorp** security policy:

```
REVOKE EXEMPTION ON RULE IDSLBACREADARRAY FOR MegaCorp FROM lynette;
```

This exemption restores the read access rules for all array components for subsequent read operations that user **lynette** attempts on tables protected by security labels of the specified policy.

When the REVOKE EXEMPTION statement successfully cancels an exemption of a user, the database server updates the **syssecpolicyexemptions** table of the system catalog to unregister the revoked exemption (or multiple exemptions, if several users are listed after the FROM keyword).

SECURITY LABEL Clause

The REVOKE SECURITY LABEL statement cancels a security label (or all the security labels of a specified security policy) held by one or more users.

SECURITY LABEL Clause

" SECURITY LABEL *policy* . { *label* | * } "

" FROM [USER] *user* "

" { FOR ALL ACCESS | [FOR READ ACCESS] | [FOR WRITE ACCESS] } "

Element	Description	Restrictions	Syntax
label	Name of an existing security label	Must exist as a label for the specified security *policy*	Identifier on page 1231
policy	The security policy of this *label*	Must already exist in the database	Identifier on page 1231

Element	Description	Restrictions	Syntax
user	User from whom the label is revoked	Must be the authorization identifier of a user	Owner name on page 1262

Only a user who holds the DBSECADM role can issue the REVOKE SECURITY LABEL statement.

A security label is a database object that is always associated with a security policy. That policy defines the set of valid security components that make up the security label. The label stores a set of one or more values for each component of the security policy.

The DBSECADM can associate a security label with the following entities:

- A column of a database table, which a *column security label* can protect
- A row of a database table, which a *row security label* can protect
- A user, whose *user security label* (and any exemptions from rules of the security policy that have been granted to the user) are called the *security credentials* of the user.

When a user who holds a security label for a specific security policy attempts to access a row that is protected by a row security label of the same security policy, the database server compares the sets of values in the user security label and in the row security label in determining whether or not the user should be allowed to access the data. Similarly, LBAC takes into account the user security label and the column security label in determining whether or not the credentials of the user should be allowed to access a protected column.

The GRANT SECURITY LABEL and REVOKE SECURITY LABEL statements enable DBSECADM to control the association of a user with a label. (Data values in a protected table are associated with a row security label or with a column security label by options to the CREATE TABLE or ALTER TABLE statements that only DBSECADM can execute, rather than by the GRANT SECURITY LABEL statement.)

Immediately following the LABEL keyword, the asterisk (*) symbol in the *policy*.* specification instructs the database server to revoke every security label of the *policy*. If instead of an asterisk you specify *policy*.*label*, that *label* must be the name of a security label of the specified *policy*. In this case, if the statement is successful, only that security label is revoked from the user list.

The USER keyword that can follow the FROM keyword is optional, but any authorization identifier specified in the REVOKE SECURITY LABEL statement must be the identifier of an individual user, rather than the identifier of a role.

Access Specifications

The list of users from whom the security label is revoked can optionally be followed by keywords that specify the type of access to data that the security policy of the label protects

- FOR WRITE ACCESS

  These keywords restrict the label to the write access rules of **IDSLBACRULES**, namely **IDLSBACWRITEARRAY**, **IDLSBACWRITESET**, and **IDLSBACWRITETREE**.

- FOR READ ACCESS

  These keywords restrict the label to the read access rules of **IDSLBACRULES**, namely **IDLSBACWREADARRAY**, **IDLSBACREADSET**, and **IDLSBACREADTREE**.

- FOR ALL ACCESS

  These keywords apply the label to all of the read and write access rules that are listed above. If the REVOKE SECURITY LABEL statement includes no FOR ... ACCESS specification, this option takes effect as the default.

For more information about these **IDSLBACRULES** rules for label-based read and write access, see Rules Associated with a Security Policy on page 301. For information about exemptions to these rules that can be granted for a specific security policy, see Rules on Which Exemptions Are Revoked on page 674.

Examples of Revoking User Security Labels

The following three statements create three security label components called **level**, **compartments**, and **groups** respectively:

```
CREATE SECURITY LABEL COMPONENT
   level ARRAY ['TS','S','C','U'];

CREATE SECURITY LABEL COMPONENT
   compartments SET {'A','B','C','D'};

CREATE SECURITY LABEL COMPONENT
   groups TREE ('G1' ROOT,
                'G2' UNDER ROOT,
                'G3' UNDER ROOT);
```

The following statement creates a security policy called **secPolicy** based on the three components above:

```
CREATE SECURITY POLICY secPolicy COMPONENTS
   level, compartments, groups;
```

The following statement creates a security label called **secLabel1**:

```
CREATE SECURITY LABEL secPolicy.secLabel1
   COMPONENT level 'S',
   COMPONENT compartments 'A', 'B',
   COMPONENT groups 'G2';
```

The following statement grants this security label for read access to user **sam**:

```
 GRANT SECURITY LABEL secPolicy.secLabel1
   TO sam FOR READ ACCESS;
```

The following statement revokes the security label for read access from user **sam**.

```
 REVOKE SECURITY LABEL secPolicy.secLabel1
   FROM sam FOR READ ACCESS;
```

When the REVOKE SECURITY LABEL statement successfully cancels a security label that was held by a user, the database server updates the **sysseclabelauth** table of the system catalog to remove the user from the list of those who hold that security label.

SETSESSIONAUTH Clause

The REVOKE SETSESSIONAUTH statement revokes the SETSESSIONAUTH privilege from one or more users or roles. The SETSESSIONAUTH privilege allows users who also hold the DBA privilege to use the SET SESSION AUTHORIZATION statement to set the session authorization to one of a set of specified users.

Syntax

SETSESSIONAUTH Clause

" **SETSESSIONAUTH ON** { **PUBLIC** | [**USER**] { *user* | "*user*" } } "

" **FROM** { | [**USER**] { *user* | "*user*" } | [**ROLE**] { *role* | "*role*" } } "

Element	Description	Restrictions	Syntax
role	Role from which the privilege is to be revoked	Must be the authorization identifier of a role	Owner name on page 1262
user	After the FROM keyword, a user from whom the privilege is to be revoked. After the ON keyword, a user whose identity the grantee can specify in the SET AUTHORIZATION statement.	Must be the authorization identifier of a user	Owner name on page 1262

Only a user who holds the DBSECADM role can revoke the SETSESSIONAUTH privilege.

The user or PUBLIC specification that follows the ON keyword specifies whose identity the grantee of the SETSESSIONAUTH privilege is no longer able to assume while using the SET SESSION AUTHORIZATION statement. This can be a user or PUBLIC, but not a role. If PUBLIC is specified, then the grantee of the privilege no longer has the ability to assume the identity of an arbitrary database user.

The USER and ROLE keywords that can follow the FROM keyword are optional. Neither the *user* nor the *role* can be the holder of the DBSECADM role who issues the REVOKE SETSESSIONAUTH statement. The FROM clause cannot specify PUBLIC.

Examples of the REVOKE SETSESSIONAUTH statement

Suppose that GRANT SETSESSIONAUTH statements issued by a user who holds the DBSECADM role had made the following assignments of the SETSESSIONAUTH privilege in the current database:

```
GRANT SETSESSIONAUTH ON lynette, manoj TO sam;
GRANT SETSESSIONAUTH ON PUBLIC TO lynette;
```

- The first example above enables user **sam** to set the session authorization to users **lynette** and **manoj**.
- The second example enables user **lynette** to set the session authorization to the PUBLIC group, or to set it to the authorization identifier of any user (but not to any role).

Both the SETSESSIONAUTH privilege and the DBA privilege are required to execute the SET AUTHORIZATION statement.

If user **sam** holds the DNA and SETSESSIONAUTH privileges, they could issue the following statement:

```
SET SESSION AUTHORIZATION TO 'lynette';
```

Now **sam** has assumed the identity of user **lynette**, including the discretionary access control (DAC) and label-based access control (LBAC) credentials of user **lynette**. User **sam** can also use this SET SESSION AUTHORIZATION statement in an API that supports Informix® trusted contexts to switch the user ID on a trusted connection. Because a previous example enabled user **lynette** to set the session authorization to any user, that is now a capability of **sam** during this session, where **sam** has assumed the identity of **lynette**.

The following example, however, revokes from user **sam** the ability to set the session authorization to users **lynette** and **manoj**:

```
REVOKE SETSESSIONAUTH ON lynette, manoj FROM sam;
```

Now the previous SET SESSION AUTHORIZATION example would fail, because this REVOKE SETSESSIONAUTH statement excludes **lynette** and **manoj** from the authorization identifiers that user **sam** can assume.

The next example revokes from user **lynette** the ability to set the session authorization to PUBLIC:

```
REVOKE SETSESSIONAUTH ON PUBLIC FROM USER lynette;
```

The PUBLIC scope of the SETSESSIONAUTH privilege that this example revokes had enabled user **lynette** (and user **sam** under the login name of **lynette** in the previous SET SESSION AUTHORIZATION example) to assume the access privileges and security credentials of any user specified by **lynette** in the SET SESSION AUTHORIZATION statement.

Delimiting *user* and *role* identifiers

If you enclose *user* or *role* in double (`"`) or single (`'`) quotation marks, the identifier is case sensitive, and the database server stores it in the system catalog exactly as you enter it in the REVOKE statement. The following REVOKE statement has the same effect as the previous example for user **lynette** whose authorization identifier includes no uppercase characters:

```
REVOKE SETSESSIONAUTH ON PUBLIC FROM USER "lynette";
```

Suppose that another user also holds the SETSESSIONAUTH privilege on the PUBLIC group, and that their authorization identifier is **Lynette**, with an initial uppercase character. Because their authorization identifier does not match the case sensitive `FROM USER "lynette"` specification, this example has no effect on their SETSESSIONAUTH privilege on the identifier of any user in the PUBLIC group. The undelimited `FROM lynette` previous example, however, revokes SETSESSIONAUTH from both **lynette** and **Lynette**.

REVOKE FRAGMENT statement

Use the REVOKE FRAGMENT statement to revoke from one or more users or roles the Insert, Update, or Delete fragment-level privileges that were granted on individual fragments of a table that has been fragmented by expression. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_938_Connect_42_refr002) unique_938_Connect_42_refr002 **REVOKE FRAGMENT** <Fragment-Level Privileges>[] **ON***table* [(*fragment*)] **FROM** { **PUBLIC** | { | *user* | *'user'* | *role* | *'role'* } } [**AS** { *revoker* | *'revoker'* }]

Element	Description	Restrictions	Syntax
fragment	Name of a fragment or the dbspace that stores one fragment. Default is all fragments of *table*.	Must exist and must store a fragment of table	Identifier on page 1231
revoker	User (who is not executing this statement) who was grantor of privileges to be revoked	Must be grantor of the fragment-level privileges	Owner name on page 1262
role	Role from which privileges are to be revoked	Must exist in the database	Owner name on page 1262
table	Fragmented table whose fragment-level privileges are to be revoked	Must exist and must be fragmented by expression	Database Object Name on page 1225
user	User from whom privileges are to be revoked	Must be a valid authorization identifier	Owner name on page 1262

Usage

The REVOKE FRAGMENT statement is a special case of the REVOKE statement for assigning privileges on table fragments. Use the REVOKE FRAGMENT statement to revoke the Insert, Update, or Delete privilege on one or more table fragments from one or more users or roles. The DBA can use this statement to revoke privileges on a fragment whose owner is another user.

The REVOKE FRAGMENT statement is valid only for tables that are fragmented by an expression-based distribution scheme. For an explanation of this fragmentation strategy, see Expression Distribution Scheme on page 35.

Specifying Fragments

If you specify no *fragment*, the privileges are revoked for all fragments of *table*. You can specify one fragment or a comma-separated list of fragments enclosed between parentheses that immediately follow the ON *table* specification.

Each *fragment* must be referenced by its name. If you did not declare an explicit identifier when you created the fragment, its name defaults to the name of the dbspace in which it resides.

After a dbspace is renamed successfully by the onspaces utility, only the new name is valid. Informix® automatically updates existing fragmentation strategies in the system catalog to substitute the new dbspace name, but you must specify the new name in REVOKE FRAGMENT statement to reference a fragment whose default name is the name of a renamed dbspace.

The FROM Clause

You can specify the PUBLIC keyword to revoke the specified fragment-level privileges from PUBLIC, thereby revoking the privileges from all users to whom the privileges have not been explicitly granted, or who do not hold a role through which they have received the privileges.

If you enclose *user* or *role* in quotation marks, the name is case sensitive and is stored exactly as you typed it. In an ANSI-compliant database, if you do not use quotation marks around *user* or around *role*, the name is stored in uppercase letters by default, although you can set the ANSIOWNER environment variable to preserve lowercase characters in *owner* specifications.

When you include a *role* in the FROM clause of REVOKE FRAGMENT, the specified fragment privilege is revoked from that role. Users who have that role, however, retain any fragment privileges they hold that were granted to them individually or to PUBLIC.

Fragment-Level Privileges

The keyword or keywords that follow the FRAGMENT keyword specify *fragment-level privileges*, which are a logical subset of table-level privileges:

Fragment-Level Privileges

```
"{ALL|{ | INSERT | DELETE | UPDATE}}"
```

You can revoke fragment-level privileges individually or in combination. The following keywords specify the fragment-level privileges that you can revoke.

Keyword

Effect

INSERT

Prevents the user from inserting rows in the fragment

DELETE

Prevents the user from deleting rows in the fragment

UPDATE

Prevents the user from updating rows in the fragment

ALL

Cancels Insert, Delete, and Update privileges on a fragment

If you specify the ALL keyword in a REVOKE FRAGMENT statement, the specified users and roles lose all fragment-level privileges that they currently possess on the specified fragments. For example, assume that a user currently has the Update privilege on one fragment of a table. If you use the ALL keyword to revoke all current privileges on this fragment from this user, the user loses the Update privilege that they had on this fragment.

For the distinction between fragment-level and table-level privileges, see the sections Definition of Fragment-Level Authorization on page 574 and Effect of Fragment-Level Authorization in Statement Validation on page 575.

The AS Clause

Without the AS clause, the user who executes the REVOKE statement must be a grantor of the privilege that is being revoked. The DBA or the owner of the fragment can use the AS clause to specify another user (who must be the grantor of the privilege) as the revoker of privileges on a fragment.

The AS clause provides the only mechanism by which privileges can be revoked on a fragment whose *owner* is an authorization identifier that is not a valid user account known to the operating system.

Examples of the REVOKE FRAGMENT Statement

Examples that follow are based on the **customer** table. They all assume that the **customer** table is fragmented by expression into three fragments named **part1**, **part2**, and **part3**.

Revoking Privileges on One Fragment

About this task

The following statement revokes the Update privilege on the fragment of the **customer** table in **part1** from user **ed**:

```
REVOKE FRAGMENT UPDATE ON customer (part1) FROM ed;
```

The following statement revokes the Update and Insert privileges on the fragment of the **customer** table in **part1** from user **susan**:

```
REVOKE FRAGMENT UPDATE, INSERT ON customer (part1) FROM susan;
```

The following statement revokes all privileges currently granted to user **harry** on the fragment of the **customer** table in **part1**:

```
REVOKE FRAGMENT ALL ON customer (part1) FROM harry;
```

Revoking Privileges on More Than One Fragment

About this task

The following statement revokes all privileges currently granted to user **millie** on the fragments of the **customer** table in **part1** and **part2**:

```
REVOKE FRAGMENT ALL ON customer (part1, part2) FROM millie;
```

Revoking Privileges from More Than One User

About this task

The following statement revokes all privileges currently granted to users **jerome** and **hilda** on the fragment of the **customer** table in **part3**:

```
REVOKE FRAGMENT ALL ON customer (part3) FROM jerome, hilda;
```

Revoking Privileges Without Specifying Fragments

About this task

The following statement revokes all current privileges from user **mel** on all fragments for which this user currently has privileges:

```
REVOKE FRAGMENT ALL ON customer FROM mel;
```

Example

Related Statements

Related statements: GRANT FRAGMENT statement on page 573 and REVOKE statement on page 654

For a discussion of fragment-level and table-level privileges, see the section Fragment-Level Privileges on page 574. See also the *IBM® Informix® Database Design and Implementation Guide*.

ROLLBACK WORK statement

Use the ROLLBACK WORK statement to cancel all or part of the current transaction intentionally, undoing any changes that occurred since the beginning of the transaction, or between the ROLLBACK WORK statement and a specified or default savepoint.

Syntax



Element	Description	Restrictions	Syntax
savepoint	Name of the savepoint that delimits the scope of the rollback	Must exist in the current transaction.	Identifier on page 1231

Usage

The ROLLBACK WORK statement is valid only in databases that support transaction logging. Only logged operations can be rolled back. Use ROLLBACK WORK only at the end of a multistatement operation.

The ROLLBACK WORK statement restores the database to its state that existed before the cancelled portion of the transaction began.

In a database that is not ANSI-compliant, the BEGIN WORK statement starts a transaction. You can end a transaction with the COMMIT WORK statement or cancel all or part of the transaction with the ROLLBACK WORK statement. If you issue the ROLLBACK WORK statement when no transaction is pending in a database that is not ANSI-compliant, Informix® issues an error.

In an ANSI-compliant database, multistatement transactions are implicit. You do not need to mark the beginning of a transaction with the BEGIN WORK statement. You only need to mark the end of each transaction with a COMMIT WORK

statement or cancel the transaction with a ROLLBACK WORK statement. If you issue the ROLLBACK WORK statement when no transaction is pending, the statement is accepted but has no effect.

The ROLLBACK WORK statement restores the database to the state that existed before the cancelled portion of the transaction began. Unless you include the TO SAVEPOINT keywords, ROLLBACK WORK cancels the entire transaction.

The ROLLBACK WORK statement releases all row and table locks that the cancelled transaction holds.

In and SPL, the ROLLBACK WORK statement closes all open cursors except those that are declared as *hold cursors* by including the WITH HOLD keywords. Hold cursors remain open after a transaction is committed or rolled back.

If you use the ROLLBACK WORK statement within an SPL routine that the WHENEVER statement calls, specify WHENEVER SQLERROR CONTINUE and WHENEVER SQLWARNING CONTINUE before the ROLLBACK WORK statement. This step prevents the program from looping if the ROLLBACK WORK statement encounters an error or a warning.

If a program terminates abnormally, the current transaction is implicitly rolled back.

WORK Keyword

The WORK keyword is optional in a ROLLBACK WORK statement. The following two statements are equivalent:

```
ROLLBACK;

ROLLBACK WORK;
```

TO SAVEPOINT Clause

The optional TO SAVEPOINT clause specifies a partial rollback. This clause can restrict the scope of the rollback to the operations of the current savepoint level between the ROLLBACK statement and the specified or default savepoint. If no *savepoint* is specified after the SAVEPOINT keyword, the rollback ends at the most recently set savepoint within the current savepoint level.

When the ROLLBACK WORK TO SAVEPOINT statement executes successfully, any effects of DDL and DML statements that preceded the savepoint persist, but changes to the schema of the database or to its data values by statements that follow the savepoint are cancelled. Any locks acquired by these cancelled statements persist, but are released at the end of the transaction. Any savepoints between the specified savepoint and the ROLLBACK statement are destroyed, but the savepoint referenced by the ROLLBACK statement (and any savepoints that precede the referenced savepoint) continue to exist. Program control passes to the statement that immediately follows the ROLLBACK statement.

If the TO SAVEPOINT clause is omitted, the ROLLBACK statement rolls back the entire transaction, and all savepoints within the transaction are released.

If the specified *savepoint* does not exist in the current transaction, the database server issues an exception.

The TO SAVEPOINT clause is not valid in a ROLLBACK statement that immediately follows the TRUNCATE statement. In this case, the attempted partial rollback fails with an error. To cancel uncommitted changes that the TRUNCATE statement has made to a table, issue ROLLBACK WORK as the next statement, but with no TO SAVEPOINT clause.

The following program fragment rolls back part of the current transaction to a savepoint called **pt109**:

```
BEGIN WORK;
DROP TABLE tab03;
CREATE TABLE tab03 (col1 CHAR(24), col2 DATE);
SAVEPOINT pt108;
...
INSERT INTO  tab03 VALUES ('First day of autumn', '09/23/2012');
SAVEPOINT pt109;
...
DELETE FROM tab03 WHERE col2 < '12/09/2009';
SAVEPOINT pt110;
...
ROLLBACK TO SAVEPOINT pt109;
```

The ROLLBACK statement in this example has these effects:

- Cancels the DML operation that deleted any rows with **col2** date values earlier than December 9, 2009.
- Releases savepoint **pt110**, and any other savepoints between **pt109** and the ROLLBACK statement.
- Cancels any other changes to the database by operations that follow savepoint **pt109** in the lexical order of SQL statements within the current transaction.

Savepoint **pt108**, however, is not released, because it was set earlier than **pt109** in the transaction. Not cancelled by this partial rollback are the effects of any uncommitted DDL or DML operations of the transaction before savepoint **pt109** was set, including the creation of table **tab03** and the INSERT operation that added a row to that table. These persist after the partial rollback, pending the possibility of another partial rollback to a savepoint, and the eventual commitment or rollback of the entire transaction.

Related Statements

Related statements: BEGIN WORK statement on page 168, COMMIT WORK statement on page 175, RELEASE SAVEPOINT statement on page 641, and SAVEPOINT statement on page 688.

For a discussion of transactions and ROLLBACK WORK, see the *HCL® Informix® Guide to SQL: Tutorial*.

SAVE EXTERNAL DIRECTIVES statement

Use the SAVE EXTERNAL DIRECTIVES statement to create external optimizer directives for a specified query, and save the directives in the database. These directives are applied automatically to subsequent instances of the same query. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

SAVE EXTERNAL DIRECTIVES *directive* { **ACTIVE** | **INACTIVE** | **TEST ONLY** } **FOR** *query*

Element	Description	Restrictions	Syntax
directive	Optimizer directive valid for *query*	Must be valid for the query and delimited by comment indicators	Optimizer Directives on page 1246

Element	Description	Restrictions	Syntax
query	Text of a valid SELECT statement	NULL string is not valid	SELECT statement on page 690

Usage

SAVE EXTERNAL DIRECTIVES associates one or more optimizer directives with a query, and stores a record of this association in the **sysdirectives** system catalog table, for subsequent use with queries that match the specified query string. This statement establishes an association between the list of optimizer directives and the text of a query, but it does not execute the specified query.

If the SAVE EXTERNAL DIRECTIVES statement specifies more than one optimizer directive, use the blank character (ASCII 32) as the separator between consecutive directives, as in the following example:

```
SAVE EXTERNAL DIRECTIVES /*+ USE_INDEX */ /*+ ORDERED */ ACTIVE FOR
     SELECT * FROM systables;
```

Unlike in a query, comma (`,`) is not a valid separator in the directives list of SAVE EXTERNAL DIRECTIVES statements.

Only the DBA or user **informix** can execute SAVE EXTERNAL DIRECTIVES. Optimizer directives that it stores in the database are called *external directives*.

External optimizer directives

External directives that the SAVE EXTERNAL DIRECTIVES statement associates with the text of a query can improve performance in some queries for which the default behavior of the query optimizer is not satisfactory.

External optimizer directives are similar to inline optimizer directives that are embedded within a query. However, unlike inline directives, external directives can be applied without revising or recompiling existing applications.

Enabling or disabling external directives for a session

Informix® ignores external directives if the EXT_DIRECTIVES parameter is set to `0` in the configuration file or the EXTDIRECTIVES keyword in the SET ENVIRONMENT statement is set to `0`, `OFF`, or `off` during a session.

In addition, the client system can disable external directives for its current session when the IFX_EXTDIRECTIVES environment variable is set to `0`.

The following table shows whether external directives are disabled (OFF) or enabled (ON) for various combinations of valid IFX_EXTDIRECTIVES settings on the client system and valid EXT_DIRECTIVES configuration parameter settings on Informix®:

Table 11. Combinations of IFX_DIRECTIVES settings and EXT_DIRECTIVES configuration parameter settings

IFX_EXTDIRECTIVES setting on client system	EXT_DIRECTIVES = 0	EXT_DIRECTIVES = 1	EXT_DIRECTIVES = 2
IFX_EXTDIRECTIVES not set	OFF	OFF	ON
IFX_EXTDIRECTIVES = 1	OFF	ON	ON

Table 11. Combinations of IFX_DIRECTIVES settings and EXT_DIRECTIVES configuration parameter settings (continued)

IFX_EXTDIRECTIVES setting on client system	EXT_DIRECTIVES = 0	EXT_DIRECTIVES = 1	EXT_DIRECTIVES = 2
IFX_EXTDIRECTIVES = 0	OFF	OFF	OFF

If EXT_DIRECTIVES is set to `1` or `2` when the database server is initialized, then the server is enabled for external directives. Individual sessions can enable or disable external directives by setting **IFX_EXTDIRECTIVES**, as the table shows. Any settings other than `1` or `2` are interpreted as zero, disabling this feature.

When external directives are enabled, the status of individual external directives is specified by the ACTIVE, INACTIVE, or TEST ONLY keywords. (But only queries on which directives are effective can benefit from external directives.)

You can also use the EXTDIRECTIVES option of the SET ENVIRONMENT statement to enable or disable external directives during a session. What you specify using the EXTDIRECTIVES option overwrites the external directive value that is specified in the EXT_DIRECTIVES configuration parameter in the ONCONFIG file.

To overwrite the value for enabling or disabling the external directive in the ONCONFIG file and:

- To enable the external directives during a session, specify `1`, `on`, or `ON` as the value for SET ENVIRONMENT EXTDIRECTIVES.
- To disable the external directives during a session, specify `0`, `off`, or `OFF` as the value for SET ENVIRONMENT EXTDIRECTIVES.

To enable the default values specified in the EXT_DIRECTIVES configuration parameter and in the client-side IFX_EXTDIRECTIVES environment variable during a session, specify DEFAULT as the value for the EXTDIRECTIVES option of the SET ENVIRONMENT statement.

For more information on using the EXTDIRECTIVES option of the SET ENVIRONMENT statement, see #unique_981.

The directive Specification

Each *directive* specification in the SAVE EXTERNAL DIRECTIVES statement must follow the syntax of the Optimizer Directives segment, as described in Optimizer Directives on page 1246, except that if you specify more than one directive, you must separate them in the directives list by a blank character, rather than by a comma (,) symbol, as in the following example:

```
SAVE EXTERNAL DIRECTIVES /*+ AVOID_INDEX (table1 index1)*/ /*+ FULL(table1) */
       ACTIVE FOR
           SELECT /*+ INDEX( table1 index1 ) */  col1, col2
               FROM table1, table2 WHERE table1.col1 = table2.col1;
```

This example associates AVOID_INDEX and FULL directives with the specified query. The inline INDEX directive is ignored by the query optimizer when the external directives are applied to a query that matches the SELECT statement.

The ACTIVE, INACTIVE, and TEST ONLY Keywords

You must include one of the ACTIVE, INACTIVE, or TEST ONLY keyword options to enable, disable, or restrict the scope of external directives:

- If external directives are enabled, the ACTIVE keyword applies the list of directives to any subsequent query that matches the *query* string.
- The INACTIVE keyword causes Informix® to ignore the directive. (It is associated with the query in **sysdirectives**, but it is dormant, with no effect.)
- If external directives are enabled, the TEST ONLY keywords apply the directives only to matching queries that the DBA or user **informix** executes. Queries by any other users cannot use TEST ONLY external directives.

An INACTIVE directive has no effect unless the DBA or user **informix** changes the **sysdirectives.active** system catalog column value from zero (INACTIVE) to one (ACTIVE) or two (TEST ONLY) for that directive. External directives do not have SQL identifiers, but the DBA can reference the **sysdirectives.id** column in an UPDATE statement to specify which external directive to update.

Alternatively, the DBA or user **informix** can delete an INACTIVE or TEST ONLY row from **sysdirectives** and use the SET EXTERNAL DIRECTIVES statement to redefine the deleted directive, but now specifying the ACTIVE keyword. This can give other users access to TEST ONLY directives that the DBA has validated.

SAVEPOINT statement

Use the SAVEPOINT statement to declare the name of a new savepoint within the current SQL transaction, and to set the position of the new savepoint within the lexical order of SQL statements within the transaction. The SAVEPOINT statement is compliant with the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_977_Connect_42_savept202) unique_977_Connect_42_savept202 **SAVEPOINT** *savepoint* [[] **UNIQUE**]

Element	Description	Restrictions	Syntax
savepoint	Name declared here for the new savepoint	Cannot be the name of an existing unique savepoint in the same savepoint level	Identifier on page 1231

Usage

You can use the SAVEPOINT statement in SQL transactions to support error handling with DB-Access and in SPL, C, and Java™ routines. You can define savepoint to partition a single complex transaction into smaller logical subsets of its component SQL statements. Within that transaction, the subset of statements that follow each savepoint can be rolled back more efficiently than if you had used multiple COMMIT WORK and ROLLBACK WORK statements in multiple transactions.

The SAVEPOINT statement sets the specified savepoint at the current position in the lexical order of statements within the current transaction. After the SAVEPOINT statement executes successfully, subsequent ROLLBACK TO SAVEPOINT

statements that reference this savepoint can cancel any uncommitted changes to the database from logged DML or DDL operations in the current transaction that follow the new savepoint but precede the ROLLBACK TO SAVEPOINT statement.

If an existing savepoint in the same transaction has the same name that the SAVEPOINT statement specifies, the existing savepoint is destroyed, unless one of the following conditions is true:

- The existing savepoint was set in a different savepoint level.
- The existing savepoint name was declared with the UNIQUE keyword option. In this case, the SAVEPOINT statement fails with an error, unless the existing UNIQUE savepoint was set in a different savepoint level.

Destroying a savepoint to reuse its name for another savepoint is not the same as releasing the savepoint. Reusing a savepoint name destroys only one savepoint. Releasing a savepoint with the RELEASE SAVEPOINT statement releases the specified savepoint and all savepoints that have been subsequently set.

The UNIQUE option

This optional keyword specifies that the application does not intend to reuse the name of this savepoint in another SAVEPOINT statement while this savepoint is active within the current savepoint level.

If a savepoint already exists that was set with the same name and with the UNIQUE keyword within the current savepoint level, the SAVEPOINT statement fails with an error, and the existing savepoint is not destroyed.

Savepoint levels

Informix® supports the construct of nested savepoint levels. A single SQL transaction can have multiple savepoint levels. New savepoint levels are automatically created for the duration of execution of an SPL routine or an external UDR. Recursive calls to the same SPL routine or UDR also increment the savepoint level of the current transaction.

A savepoint level ends when the UDR in which it was created finishes execution. When a savepoint level ends, all savepoints within it are automatically released. Any DDL or DML modifications are inherited by the parent savepoint level (that is, by the savepoint level within which the one that just ended was created), and are subject to any savepoint-related statements that are issued against the parent savepoint level.

The following rules apply to actions within a savepoint level:

- Savepoints can only be referenced within the savepoint level in which they are established. You cannot release, destroy, or roll back to a savepoint established outside of the current savepoint level.
- The uniqueness of savepoint names is only enforced within the current savepoint level. The names of savepoints that are active in other savepoint levels can be reused in the current savepoint level without affecting those savepoints in other savepoint levels.

Savepoints in distributed SQL transactions

Savepoints are valid in cross-database distributed SQL transactions of a single Informix® instance that supports transactions if all of the participating databases support transaction logging. Savepoints are also supported in cross-server

SQL transactions, including operations in high-availability clusters, if all of the participating Informix® instances support savepoints, and all of the databases that are accessed in the transaction use logging.

If any of the participating database servers in a cross-server transaction does not support savepoints, however, and a connection is established between a coordinator that can support savepoints and a subordinate server that cannot, any ROLLBACK TO SAVEPOINT statement within the distributed transaction fails with an error.

Persistence of savepoints

Savepoints are position markers within SQL transactions, not database objects. An existing savepoint **S** is destroyed by any of the following events within the same transaction:

- A COMMIT WORK or ROLLBACK WORK (without the TO SAVEPOINT clause) statement is executed.
- A RELEASE SAVEPOINT statement is executed that specifies **S** in the same savepoint level.
- A ROLLBACK TO SAVEPOINT or RELEASE SAVEPOINT statement is executed that specifies a savepoint that was established earlier than **S** in the same savepoint level.
- A SAVEPOINT statement specifies the same name as **S** in the same savepoint level, and **S** was not created with the UNIQUE keyword.

Restrictions on savepoints

Savepoints and savepoint levels are not supported in the following contexts:

- in databases that do not support transaction logging
- in triggered actions
- in XA global transactions
- in applications or UDRs where the AUTOCOMMIT connection attribute is enabled.

In addition, the SAVEPOINT statement (like the RELEASE SAVEPOINT and ROLLBACK WORK TO SAVEPOINT statements) is not valid in UDRs that are invoked within DML statements, as in the following example:

```
SELECT first_1 foo() FROM systables;
```

Here the **foo( )** routine cannot set a savepoint.

Related Statements

Related statements: COMMIT WORK statement on page 175, RELEASE SAVEPOINT statement on page 641, and ROLLBACK WORK statement on page 683

SELECT statement

Use the SELECT statement to retrieve values from a database or from an SPL or collection variable. A SELECT operation is called a *query*.

Rows or values that satisfy the specified search criteria of the query are called *qualifying* rows or values. What the query retrieves to its calling context, after applying any additional logical conditions, is called the *result set* of the query. This result set can be empty.

Syntax

(explicit id unique_471_Connect_42_sel003) unique_471_Connect_42_sel003(explicit id unique_471_Connect_42_sel004)

unique_471_Connect_42_sel004(explicit id unique_471_Connect_42_sel006) unique_471_Connect_42_sel006(explicit id unique_471_Connect_42_sel007)

unique_471_Connect_42_sel007(explicit id unique_471_Connect_42_sel008) unique_471_Connect_42_sel008(explicit id unique_471_Connect_42_sel009)

unique_471_Connect_42_sel009(explicit id unique_471_Connect_42_sel010) unique_471_Connect_42_sel010(explicit id unique_471_Connect_42_sel011)

unique_471_Connect_42_sel011(explicit id unique_471_Connect_42_sel012) unique_471_Connect_42_sel012(explicit id unique_471_Connect_42_sel013)

unique_471_Connect_42_sel013(explicit id unique_471_Connect_42_sel014) unique_471_Connect_42_sel014(explicit id unique_471_Connect_42_sel015)

unique_471_Connect_42_sel015(explicit id unique_471_Connect_42_sel016) unique_471_Connect_42_sel016(explicit id unique_471_Connect_42_sel017)

unique_471_Connect_42_sel017(explicit id unique_471_Connect_42_sel018) unique_471_Connect_42_sel018(explicit id unique_471_Connect_42_sel019)

unique_471_Connect_42_sel019(explicit id unique_471_Connect_42_sel020) unique_471_Connect_42_sel020(explicit id unique_471_Connect_42_sel021)

unique_471_Connect_42_sel021 **SELECT** <Select Options> [{ **UNION [ALL]** | **INTERSECT** | { **MINUS** | **EXCEPT** } }] **SELECT** <Select Options>] [<ORDER BY Clause>] [<LIMIT Clause>] [[{ **FOR READ ONLY** | **FOR UPDATE** [**OF** *column*] }] [<INTO Table Clause>]

Select options

“ [<Optimizer Directives>] ”

“ <Projection Clause> ”

“ [<INTO Clause>] ”

“ <FROM Clause> ”

“ [<GRID Clause>] ”

“ [<WHERE Clause>] [<Hierarchical Clause>] ”

“ [<GROUP BY Clause>] ”

“ [<HAVING Clause>] ”

Element	Description	Restrictions	Syntax
column	Name of a column that can be updated after a FETCH	Must be in a FROM clause table, but does not need to be in the select list of the Projection clause	Identifier on page 1231

Usage

The SELECT statement can return data from tables in the current database, or in another database of the current database server, or in a database of another database server. Only the SELECT keyword, the Projection clause, and the FROM clause are required specifications.

For hierarchical queries that include the CONNECT BY clause, the FROM clause can specify only a single table that must reside in the local database of the Informix® database server instance to which the current session is connected.

For queries that include the GRID clause, the instances of each table that the FROM clause specifies must have the same schema, the same database locale, and the same code set on every node that the GRID clause specifies.

The SELECT statement can reference no more than one external table that the CREATE EXTERNAL TABLE statement has defined. In complex queries, this external table can be specified only in the outermost query. You cannot reference an external table in a subquery.

You need the Connect access privilege on the database to execute a query, as well as the Select privilege on the table objects from which the query retrieves rows.

The SELECT statement can include various basic clauses, which are identified in the following list.

Clause	Effect
Optimizer Directives on page 1246	Specifies how the query should be implemented
Projection Clause on page 694	Specifies a list of items to be read from the database
INTO Clause on page 709	Specifies variables to receive the result set
FROM Clause on page 711	Specifies the data sources of Projection clause items
Aliases for Tables or Views on page 713	Temporary names for tables or columns in a query
Table expressions on page 714	Define derived tables as query data sources
Lateral derived tables on page 715	Define correlated table references in a query
The ONLY Keyword on page 718	Excludes child tables as data sources in queries of typed tables
Iterator Functions on page 720	Functions repeatedly returning values as a data source
ANSI Joins on page 721	Join queries compliant with ISO/ANSI syntax standards

Clause	Effect
Informix-Extension Outer Joins on page 726	Query syntax based on implicit LEFT OUTER joins
GRID clause on page 728	Specifies the nodes that store the tables of a grid query
Using the ON Clause on page 724	Specifies join conditions as pre-join filters
WHERE clause of SELECT on page 731	Sets conditions on qualifying rows and post-join filters
Hierarchical Clause on page 738	Sets conditions for queries of hierarchical data
GROUP BY Clause on page 752	Combines groups of rows into summary results
HAVING Clause on page 754	Sets conditions on the summary results
ORDER BY Clause on page 760	Sorts qualifying rows according to column values
ORDER SIBLINGS BY Clause on page 766	Sorts hierarchical data for siblings at every level
LIMIT Clause on page 768	Limits how many qualifying rows can be returned
FOR UPDATE Clause on page 771	Enables updating of the result set after a FETCH
FOR READ ONLY Clause on page 773	Disables updating of the result set after a FETCH
INTO TEMP clause	Puts the result set into a temporary table
INTO EXTERNAL clause	Stores the query result set in an external table
INTO STANDARD and INTO RAW Clauses	Stores the query result set in a permanent database table
UNION Operator on page 758	Combines the result sets of two SELECT statements and optionally discards duplicate rows
INTERSECT Operator on page 759	Returns distinct common rows from two query result sets
MINUS operator on page 760	Returns distinct rows that only the first of two queries return.

Projection Clause

The Projection clause (sometimes called the *Select clause*) specifies a list of database objects or expressions to retrieve, and can set restrictions on qualifying rows. (The *select list* is sometimes also called the *projection list*.)

This syntax fragment is part of the SELECT statement on page 690.

(explicit id unique_985_Connect_42_procl004) unique_985_Connect_42_procl004(explicit id unique_985_Connect_42_procl008) unique_985_Connect_42_procl008(explicit id unique_985_Connect_42_procl006) unique_985_Connect_42_procl006(explicit id unique_985_Connect_42_procl007) unique_985_Connect_42_procl007(explicit id unique_985_Connect_42_procl018) unique_985_Connect_42_procl018(explicit id unique_985_Connect_42_procl009) unique_985_Connect_42_procl009(explicit id unique_985_Connect_42_procl010) unique_985_Connect_42_procl010

Projection Clause

" [**SKIP** { *offset* | *off_var* } []] "

" [{ { **FIRST** | **LIMIT** } } { *max* | *max_var* } []] "

" { **ALL** | [{ **DISTINCT** | [] **UNIQUE** }] [] } "

" `<Select List>` "

Select List

" { { `<OLAP Window expression>` [] | `<Expression>` [] | *column* } [[**AS**] *display_label*] | [[{ *table.* | *view.* | *synonym.* | *alias.* }] { *column* [[**AS**] { *column_alias* | *display_label* }] | `*` }] | { *external.* `*` | ({ `<Collection Subquery>` [] | *subquery* }) } } "

Element	Description	Restrictions	Syntax
alias	Temporary *table* or *view* name. See Declaring a Column Alias on page 709.	Valid only if the FROM clause declares the *alias* for *table* or *view*. See FROM Clause on page 711.	Identifier on page 1231
column_alias	Temporary identifier that you declare here for a *column*	Must be unique among *columns* and *column_alias* names in this query. Only the GROUP BY clause can reference a *column_alias*.	Identifier on page 1231
column	Column from which to retrieve data	Must exist in a data source that the FROM clause references	Identifier on page 1231
display_label	Temporary name declared here for a *column* or for an expression	See Declaring a Display Label on page 708	Identifier on page 1231

Element	Description	Restrictions	Syntax
external	External table from which to retrieve data	Must exist	Database Object Name on page 1225
max	Integer (> 0) specifying maximum number of rows to return	If *max* > number of qualifying rows then all matching rows are returned	Literal Number on page 1198
max_var	Host variable or local SPL variable storing the value of *max*	Same as *max*; valid in prepared objects and in SPL routines	Language dependent
offset	Integer (> 0) specifying how many qualifying rows to exclude before the first row of the result set	Cannot be negative. If *offset* > (number of qualifying rows), then no rows are returned	Literal Number on page 1198
off_var	Host variable or local SPL variable storing the value of offset	Same as *offset*; valid in prepared objects and in user-defined routines	Language dependent
subquery	Embedded query	A subquery within the Projection clause cannot include the SKIP, FIRST, INTO TEMP, or the ORDER BY clause.	SELECT statement on page 690
table, *view*, *synonym*	Name of a table, view, or synonym from which to retrieve data	The synonym and the table or view to which it points must exist	Database Object Name on page 1225

The asterisk (*) specifies all columns in the *table* or *view* in their defined order. To retrieve all columns in another order, or a subset of columns, you must specify individual *column* names explicitly. A solitary asterisk (*) can be a valid Projection clause if the FROM clause specifies only a single data source.

To execute a query, the database server constructs a query plan and retrieves all qualifying rows that match the WHERE clause conditions. (Here a *row* refers to one set of values, as specified in the select list, from a single record in the table or joined tables that the FROM clause specifies.) If the query has no ORDER BY clause, the qualifying rows are sequenced in the order of their retrieval, which might vary with each execution; otherwise, their sequence follows the ORDER BY specification.

Restricting return values with the SKIP, LIMIT, and FIRST options

You can use the SKIP, LIMIT, and FIRST options to restrict the number of return values. The SKIP option excludes the first specified number of return values. The LIMIT and FIRST options set the maximum number of return values.

There is no default value for the *max* or *offset* parameters of the FIRST, LIMIT, or SKIP options. If no integer or integer variable follows the FIRST, LIMIT, or SKIP keyword, the database server interprets that keyword as a column identifier. If no data source in the FROM clause has a column with that name, the query fails with an error.

You can use these features in cross-server distributed queries only if all of the participating database server instances support the SKIP, LIMIT, and FIRST options.

SKIP option

The SKIP *offset* option specifies the number of qualifying rows to exclude, for *offset* an integer in the SERIAL8 range, counting from the first qualifying row.

The following example retrieves the values from all rows except the first 10 rows:

```
SELECT SKIP 10 a, b FROM tab1;
```

You can also use a host variable to specify how many rows to exclude. In an SPL routine, you can use an input parameter or a local variable to provide this value.

You can also use the SKIP option to restrict the result sets of prepared SELECT statements, of UNION queries, in queries whose result set defines a collection-derived table, and in the events and actions of triggers.

The SKIP option is not valid in the following contexts:

- In the definition of a view
- In nested SELECT statements
- In subqueries, except for table expressions in the FROM clause.

FIRST option

The FIRST *max* option specifies that the result set includes no more than *max* rows (or exactly *max*, if *max* is not greater than the number of qualifying rows). Any additional rows that satisfy the selection criteria are not returned. If there are fewer than *max* qualifying rows, the sorted and unsorted query results always include the same rows, but typically not in the same order.

The following example retrieves at most 10 rows from table **tab1**:

```
SELECT FIRST 10 a, b FROM tab1;
```

You can use a host variable or the value of an SPL input parameter in a local variable to specify *max*.

The FIRST keyword is also valid in queries that include table expressions in the FROM clause. The following example returns the first 8 qualifying rows, in their order of retrieval from the table expression in the FROM clause of the outer query:

```
SELECT * FROM (SELECT FIRST 8 col1
                  FROM tab1 WHERE col1 > 50 );
```

The next example applies the FIRST option to the result of a UNION expression:

```
SELECT FIRST 10 a, b FROM tab1 UNION SELECT a, b FROM tab2;
```

The FIRST option is not valid in any of the following contexts:

- In the definition of a view
- In nested SELECT statements
- In subqueries, except for subqueries that specify table expressions in the FROM clause
- In a singleton SELECT (where *max* = 1) within an SPL routine
- Where embedded SELECT statements are used as expressions

Instead of the FIRST or LIMIT option in the Projection clause, you can instead include the LIMIT clause after the Projection clause and the optional ORDER BY clause. For more information on the LIMIT clause, see LIMIT Clause on page 768.

LIMIT option

LIMIT is a keyword synonym for the FIRST keyword in the Projection clause. However, you cannot substitute LIMIT for FIRST in other syntactic contexts where FIRST is valid, such as in the FETCH statement.

Combining the SKIP option and the FIRST or LIMIT option

If a Projection clause with the SKIP *offset* option also includes a *max* limit that the FIRST or LIMIT option specifies, the query result set begins with the row whose ordinal position among qualifying rows is (*offset* + 1), rather than with the first row that satisfies the selection criteria.

If no ORDER BY clause sorts the retrieved rows, the row in position (*offset* + *max*) is the last row in the result set, unless there are fewer than (*offset* + *max*) qualifying rows. The following example ignores the first 50 qualifying rows in table **tab1**, but returns a result set of at most 10 rows, beginning with the fifty-first row:

```
SELECT SKIP 50 FIRST 10 a, b FROM tab1;
```

The SELECT clause in the next example uses the SKIP and FIRST options to insert no more than five rows from table **tab1** into table **tab2**, beginning with the eleventh row:

```
INSERT INTO tab2 SELECT SKIP 10 FIRST 5 * FROM tab1;
```

Sorting results from SKIP, LIMIT, or FIRST queries

When you combine the ORDER BY clause with the SKIP, LIMIT, or FIRST options in a query, the ORDER BY clause can affect which qualifying rows are in the result set.

Queries that combine the SKIP and the FIRST or LIMIT options with the ORDER BY clause impose a unique order on the qualifying rows. Successive queries that increment the *offset* value by the value of *max* can partition the results into distinct subsets of *max* rows. This capability can support web applications that require a fixed page size, without requiring cursor management.

Using FIRST or LIMIT with ORDER BY

For a query that includes the FIRST or LIMIT optionor the LIMIT clause and an ORDER BY clause, the qualifying rows that are returned in their sort-key order might not be the same result set that the query without ORDER BY returned in the order of retrieval.

With an ORDER BY clause, you can retrieve the first *max* qualifying rows. For example, the following query finds the ten highest-paid employees:

```
SELECT FIRST 10 name, salary FROM emp ORDER BY salary DESC;
```

You can use the FIRST option in a query whose result set defines collection-derived table (CDT) within the FROM clause of another SELECT statement. The following query specifies a CDT that has no more than ten rows:

```
SELECT *
   FROM TABLE(MULTISET(SELECT FIRST 10 * FROM employees
   ORDER BY employee_id)) vt(x,y), tab2
   WHERE tab2.id = vt.x;
```

Using SKIP and ORDER BY

For a query in which the SKIP option defines an integer *offset* of qualifying rows that are ignored before the first returned row, the order of retrieval determines which rows are omitted from the query result if the ORDER BY clause is absent. However, if the ORDER BY clause is included the *offset* is applied to the sorted rows. Whether sorted in ascending or descending order, these rows that are excluded based on their sort-key value are generally uncorrelated with the qualifying rows that are excluded based on their order of retrieval, unless the query result set is empty.

When you use the SKIP option in a query with an ORDER BY clause, you can exclude the first *offset* rows that have the lowest values according to the ORDER BY criteria. You can also use SKIP to exclude rows with the highest values, if the ORDER BY clause includes the DESC keyword. For example, the following query returns all rows of the **orders** table, except for the fifty oldest orders:

```
SELECT SKIP 50 * FROM orders ORDER BY order_date;
```

Here the result set is empty if there are fewer than 50 rows in the **orders** table. An *offset* of zero (`0`) is not invalid, but in that case the SKIP option does nothing.

The following example query contains a collection subquery that performs the following actions:

- retrieves only the eleventh through fifteenth rows from **tab3** as a collection-derived table,
- sorts those five rows by the ascending order of their value in column **a**,
- and stores this result set in a temporary table.

```
SELECT * FROM TABLE (MULTISET (SELECT SKIP 10 FIRST 5 a FROM tab3
                     ORDER BY a)) INTO TEMP;
```

The following INSERT statement includes a collection subquery whose results define a collection-derived table. The rows are ordered by the value in column **a**, and are inserted into table **tab1**.

```
INSERT INTO tab1 (a) SELECT *
   FROM TABLE (MULTISET (SELECT SKIP 10 FIRST 5 a
                            FROM tab3 ORDER BY a));
```

Controlling duplicate return values

You can apply the ALL, UNIQUE, or DISTINCT keywords to indicate whether duplicate values are returned, if any exist. If you do not specify any of these keywords in the Projection clause, all qualifying rows are returned by default.

ALL

Default. Specifies that all qualifying rows are returned, regardless of whether duplicates exist.

DISTINCT

Excludes duplicates of qualifying rows from the result set

UNIQUE

Excludes duplicate. UNIQUE is a synonym for DISTINCT, as an extension to the ANSI/ISO standard.

For example, the next query returns all the unique ordered pairs of values from the **stock_num** and **manu_code** columns in rows of the **items** table. If several rows have the same pair of values, that pair appears only once in the result set:

```
SELECT DISTINCT stock_num, manu_code FROM items;
```

For information on how the database server identifies duplicate NCHAR and NVARCHAR values in a database that has the NLCASE INSENSITIVE property, see NCHAR and NVARCHAR expressions in case-insensitive databases.

You can specify the DISTINCT or UNIQUE keyword of the SELECT statement no more than once in each level of a query or subquery. The following example uses DISTINCT in both the query and in the subquery:

```
SELECT DISTINCT stock_num, manu_code FROM items
   WHERE order_num = (SELECT DISTINCT order_num FROM orders
      WHERE customer_num = 120);
```

The example above is valid, because DISTINCT is used no more than once in each of the SELECT statements.

If a query includes the DISTINCT or UNIQUE keyword (rather than the ALL keyword or no keyword) in the Projection clause whose Select list also includes an aggregate function whose argument list begins with the DISTINCT or UNIQUE keyword, the database server issues an error, as in the following example: .

```
SELECT DISTINCT COUNT(DISTINCT ship_weight)
       FROM orders;
```

That is, it is not valid in the same query for both the Projection clause and for an aggregate function to restrict the result set to unique values. (In the example above, replacing either of the DISTINCT keyword with UNIQUE would not avoid this error.)

Queries with multiple aggregate expressions

If the Projection clause does not specify the DISTINCT or UNIQUE keyword of the SELECT statement, the query can include multiple built-in aggregate functions that each includes the DISTINCT or UNIQUE keyword as the first specification in the argument list, as in the following example:

```
SELECT COUNT (DISTINCT customer_num),
       COUNT (UNIQUE order_num),
       AVG(DISTINCT ship_charge) FROM orders;
```

Support for multiple DISTINCT or UNIQUE aggregate expressions in the same level of a query applies to built-in aggregate functions, but not to user-defined aggregate (UDA) functions that the CREATE AGGREGATE statement defined. The database server issues an error if the argument lists of more than one UDA expression in the same query begin with the DISTINCT or UNIQUE keywords.

Duplicate rows in NLSCASE INSENSITIVE databases

In a database that was created with the NLSCASE INSENSITIVE option, columns and expressions of NCHAR or NVARCHAR data types make no distinction between upper case and lower case letters, so that strings of these data types that have the same sequence of characters, but with letter case variants, evaluate as duplicates.

Queries that include the ALL, DISTINCT, or UNIQUE keywords might return results different from what the same query returns from a case-sensitive database into which the same character string values had been loaded. For example, the NVARCHAR strings `"aCe"`, `"ACE"`, and `"AcE"` evaluate as identical in databases that have the NLSCASE INSENSITIVE property, but the same three strings are processed as distinct values in case-sensitive databases.

Strings of type CHAR, LVARCHAR, and VARCHAR, however, are processed identically in NLSCASE SENSITIVE and in NLSCASE INSENSITIVE databases by queries that use the ALL, DISTINCT, or UNIQUE keywords to include or exclude duplicate rows. For more information about databases with the NLSCASE INSENSITIVE property, see Specifying NLSCASE case sensitivity on page 195 and NCHAR and NVARCHAR expressions in case-insensitive databases.

Data Types in Distributed Queries

Queries whose only data sources are tables and views in the local database to which the session is connected can return values from columns or expressions of any built-in or user-defined data type that is registered in the local database. Queries that reference tables or views in other databases are called *distributed queries*, and the data types that they can access are a subset of the data types that Informix® supports in local queries.

Among distributed queries, the restrictions on data types depend on the number of participating database servers.

- If all the databases that the query accesses are databases of the same Informix® instance, the query is called a *cross-database* distributed query.
- If the query accesses databases of multiple Informix® instances, the query is called a *cross-server* distributed query.

In both types of distributed queries, all participating databases must have the same ANSI/ISO-compliance status. A cross-server distributed query can use both the SKIP and FIRST options if all participating servers support the SKIP option; otherwise the query fails with an error. More generally, all cross-server operations require that the participating database server instances support the SQL syntax that specifies the operation.

For additional information about distributed queries, see the *IBM® Informix® Database Design and Implementation Guide*.

Data Types in Cross-Database Transactions

A query that accesses tables in more than one database of the local server instance can return only a subset of the SQL data types that a local query can return from tables in the local database. Similarly, a cross-server distributed query on tables in databases of more than one server instance has additional restrictions on returned data types.

Distributed queries (and other distributed DML operations or function calls) that access only databases of the local Informix® instance can access data types of the following categories:

- The *built-in data types* that are not opaque, including these:
  - BIGINT
  - BIGSERIAL
  - BYTE
  - CHAR
  - DATE

    - DATETIME
    - DECIMAL
    - FLOAT
    - INT
    - INTERVAL
    - INT8
    - MONEY
    - NCHAR
    - NVARCHAR
    - SERIAL
    - SERIAL8
    - SMALLFLOAT
    - SMALLINT
    - TEXT
    - VARCHAR
- Most *built-in opaque data types*, including these:
    - BLOB
    - BSON
    - BOOLEAN
    - CLIENTBINVAL
    - CLOB
    - IFX_LO_SPEC
    - IFX_LO_STAT
    - INDEXKEYARRAY
    - JSON
    - LVARCHAR
    - POINTER
    - RTNPARAMTYPES,
    - SELFUNCARGS
    - STAT
    - XID
- User-defined types (UDTs) that are cast explicitly to any of the built-in types that are listed above
- DISTINCT of any of the built-in types in the preceding list.

Distributed operations across databases of the local Informix® instance can return UDTs and DISTINCT types based on built-in data types only if all the UDTs and DISTINCT types are cast explicitly to built-in data types.

All the opaque UDTs, DISTINCT types, data type hierarchies, and casts must have exactly the same definitions in each database that participates in the distributed query. For queries or other DML operations in cross-database UDRs that use the data types listed above as parameters or as returned data types, the UDR must also have the same definition in each participating database.

A cross-database distributed query (or any other cross-database DML operation) fails with an error if it references a table, view, or synonym in another database of the local Informix® instance that includes a column of any of the following data types:

- LOLIST
- IMPEXP
- IMPEXPBIN
- SENDRECV
- DISTINCT of any of the built-in opaque data types that are listed above.
- Complex types (named or unnamed ROW, COLLECTION, LIST, MULTISET, or SET)
- DISTINCT of ROW.

Data Types in Cross-Server Transactions

A distributed query (or any other distributed DML operation or function call) across databases of two or more Informix® instances cannot return complex or large-object data types, nor most user-defined data types (UDTs) or opaque data types.

A distributed query (or any other distributed DML operation or function call) across databases of two or more Informix® instances cannot return complex or large-object data types, nor most user-defined data types (UDTs) or opaque data types. Cross-server distributed queries, DML operations, and function calls can return only the following atomic data types, where *atomic* excludes the complex data types:

- Any non-opaque atomic built-in data type
- BOOLEAN
- BSON
- JSON
- LVARCHAR
- DISTINCT of non-opaque built-in types
- DISTINCT of BOOLEAN
- DISTINCT of BSON
- DISTINCT of JSON
- DISTINCT of LVARCHAR
- DISTINCT of any of the DISTINCT types that appear above in this list.

A cross-server distributed query can support DISTINCT data types only if both of the following conditions are true for every DISTINCT type in the query:

- The DISTINCT data type is explicitly cast to one of the atomic built-in types above,
- The DISTINCT type, its data type hierarchy, and its casts are defined exactly the same way in each database that participates in the distributed query.

For queries or other DML operations in cross-server UDRs (user-defined routines) that use the data types in the preceding list as parameters or as returned data types, the UDR must also have the same definition in every participating database.

Cross-server queries of protected tables

The built-in DISTINCT data type IDSSECURITYLABEL, which stores security label objects, can be accessed in cross-server and cross-database operations on protected data by users who hold sufficient security credentials for the same label-based access control (LBAC) security policy. Like local operations on protected data, distributed queries that access remote tables protected by a security policy can return only the qualifying rows that **IDSLBACRULES** allow, after the database server has compared the security label that secures the data with the security credentials of the user who issues the query.

For additional information about the data types that Informix® supports in cross-server DML operations, see Data Types in Distributed Queries on page 700. For information about the table hierarchies of the DISTINCT data types that are valid in cross-server operations, see DISTINCT Types in Distributed Operations on page 996.

Built-in data types not valid in cross-server queries

A cross-server query (or any other cross-server DML operation) fails with an error if it references a table, view, or synonym in a database of another Informix® instance that includes a column of any of the following data types:

- BLOB
- BYTE
- CLIENTBINVAL
- CLOB
- IFX_LO_SPEC
- IFX_LO_STAT
- INDEXKEYARRAY
- POINTER
- RTNPARAMTYPES
- SELFUNCARGS
- STAT
- TEXT
- XID
- User-defined OPAQUE types
- Complex types (named or unnamed ROW, COLLECTION, LIST, MULTISET, or SET)
- DISTINCT of any of the opaque or complex data types that are listed above.

Requirements for participating database servers and databases

Cross-server operations require that all participating database server instances support the SQL syntax that specifies the operation.

Cross-server queries cannot access the database of another Informix® instance unless both servers define TCP/IP or IPCSTR connections in their DBSERVERNAME or DSERVERALIASES configuration parameters and in the **sqlhosts** information. The requirement that both participating servers support the same type of connection (either TCP/IP or else IPCSTR) applies to any communication between Informix® instances, even if both reside on the same computer.

For the SQL syntax to reference objects in databases of remote server instances that are not part of an Informix® grid, see Specifying a Database Object in a Cross-Server Query on page 1227. For information about grid queries, see GRID clause on page 728.

Expressions in the Select List

You can use any basic type of expression (column, constant, built-in function, aggregate function, and user-defined routine), or combination thereof, in the select list. The expression types are described in Expression on page 1003. Sections that follow present examples of simple expression in the select list.

You can combine simple numeric expressions by connecting them with arithmetic operators for addition, subtraction, multiplication, and division. If you combine a column expression and an aggregate function, however, you must include the column expression in the GROUP BY clause. (See also Dependencies between the GROUP BY and Projection clauses on page 753.)

In general, you cannot use variables (for example, host variables in an ESQL/C application) in the select list by themselves. A variable is valid in the select list, however, if an arithmetic or concatenation operator connects it to a constant.

In a FOREACH SELECT statement, you cannot use SPL variables in the select list, by themselves or with column names, when the tables in the FROM clause are remote tables. You can use SPL variables by themselves or with a constant in the select list only when the tables in the FROM clause are local tables.

In distributed queries of Informix®, values in expressions (and returned by expressions) are restricted, as Data Types in Cross-Server Transactions on page 702 describes. Any UDRs whose return values are used as expressions in other databases of the same Informix® instance must be defined in each participating database.

The Boolean operator NOT is not valid in the Projection clause.

Selecting Columns

Column expressions are the most commonly used expressions in a SELECT statement. For a complete description of the syntax and use of column expressions, see Column Expressions on page 1025. The following examples use column expressions in the Projection clause:

```
SELECT orders.order_num, items.price FROM orders, items;
SELECT customer.customer_num ccnum, company FROM customer;
SELECT catalog_num, stock_num, cat_advert [1,15] FROM catalog;
SELECT lead_time - 2 UNITS DAY FROM manufact;
```

Selecting Constants

If you include a constant expression in the projection list, the same value is returned for each row that the query returns (except when the constant expression is **NEXTVAL**). For a complete description of the syntax and use of constant expressions, see Constant Expressions on page 1040. Examples that follow show constant expressions within a select list:

```
SELECT 'The first name is', fname FROM customer;
SELECT TODAY FROM cust_calls;
SELECT SITENAME FROM systables WHERE tabid = ;1
```

```
SELECT lead_time - 2 UNITS DAY FROM manufact;
SELECT customer_num + LENGTH('string') from customer;
```

Selecting Built-In Function Expressions

A built-in function expression uses a function that is evaluated for each row in the query. All built-in function expressions require arguments. This set of expressions contains the time functions and the length function when they are used with a column name as an argument. The following examples show built-in function expressions within the select list of the Projection clause:

```
SELECT EXTEND(res_dtime, YEAR TO SECOND) FROM cust_calls;
SELECT LENGTH(fname) + LENGTH(lname) FROM customer;
SELECT HEX(order_num) FROM orders;
SELECT MONTH(order_date) FROM orders;
```

Selecting Aggregate Function Expressions

An aggregate function returns one value for a set of queried rows. This value depends on the set of rows that the WHERE clause of the SELECT statement qualifies. In the absence of a WHERE clause, the aggregate functions take on values that depend on all the rows that the FROM clause forms.

Examples that follow show aggregate functions in a projection list:

```
SELECT SUM(total_price) FROM items WHERE order_num = 1013;
SELECT COUNT(*) FROM orders WHERE order_num = 1001;
SELECT MAX(LENGTH(fname) + LENGTH(lname)) FROM customer;
```

If the Projection clause does not specify the DISTINCT or UNIQUE keyword of the SELECT statement, however, the query can include one or more aggregate functions that include the DISTINCT or UNIQUE keyword as the first specification of the argument lists:

```
SELECT SUM(DISTINCT total_price) FROM items WHERE order_num = 1013;
SELECT COUNT(DISTINCT *) FROM orders WHERE order_num = 1001;
SELECT MAX(LENGTH(fname) + LENGTH(UNIQUE lname)) FROM customer;
```

The database server issues an error, however, if both the Projection clause and an aggregate function expression specify the DISTINCT or UNIQUE keyword in the same query.

Aggregate expressions in grid queries

For grid queries that include aggregate function expressions, you must specify the GRID clause in a subquery, if the value of the aggregate expression that each grid server calculates is a quotient whose denominator varies across the participating grid servers.

SQL aggregate expressions and OLAP window aggregates

Do not confuse SQL aggregate functions with the On-Line Analytical Processing (OLAP) window aggregation functions, which are a different category of functions.

When an aggregate function expression is immediately followed by the OVER clause, the database server attempts to interpret it as an OLAP aggregation function. Some OLAP aggregation functions have the same names (and support a subset of the same syntax) as SQL aggregate functions, but the two types of functions have different behavior.

An SQL aggregate function can be nested inside an OLAP aggregation function. For example, the following query is valid in a context where **dollars** is a column in the sales **table**:

```
SELECT AVG(SUM(dollars)) OVER() FROM sales;
```

In the example above, the SUM function is an SQL aggregate function and the containing AVG function is an OLAP window function. The order of query processing dictates that OLAP functions are always calculated after grouping and aggregation operations but before the final ORDER BY operation.

Selecting OLAP window expressions

You can include OLAP window expressions in the select list of the Projection clause.

The On-Line Analytical Processing (OLAP) functions can return ranking, row numbering, and aggregate function information for the entire result set of a query or subquery, or for partitioned subsets of the qualifying rows that the OLAP window defines. You can use OLAP specifications to define moving windows within a partition of the result set for examining dimensions of the data, and identifying patterns, trends, and exceptions within data sets.

A query that includes OLAP window expressions returns the rows in the result set of the query, and the results of the OLAP window functions, if those functions return anything.

An OLAP window aggregate function expression can be the argument to another OLAP window aggregate function. OLAP window aggregates cannot, however, be arguments to non-analytic aggregate functions.

Selecting correlated aggregates in subqueries

In a subquery, an aggregate expression with a column operand that was declared in a parent query block is called a *correlated aggregate*. The column operand is called a *correlated column reference*.

SELECT statements and other DML statements can include subqueries with aggregate expressions whose operands reference columns that were declared in a parent query block. When a subquery contains an aggregate with a correlated column reference, the database server evaluates that aggregate in the parent query block where the correlated column reference was declared.

If an aggregate expression in a subquery contains correlated references to columns from more than one parent query block, the correlated aggregate is evaluated in the parent that is nearest to the subquery in the lexical order of query blocks.

Examples of aggregates with correlated column references

In the following example, the database server evaluates the correlated aggregate **`COUNT(n.j)`** in the parent query block, which declared the table alias **`n`** that appears in the correlated column reference of the subquery:

```
CREATE TABLE tab(i INT);
CREATE TABLE tab2(j INT);
. . .
SELECT m.i,
```

```
       (SELECT COUNT(n.j)
           FROM tab2 WHERE j=15) AS o
    FROM tab m, tab2 n GROUP BY 1;
```

The only exception to the rule illustrated above occurs when the correlated aggregate is an argument to an aggregate in the parent query block. In this scenario, the aggregate is evaluated in the subquery, if the correlated column reference comes from the same query block where the outer aggregate is specified. The next example illustrates nested aggregates:

```
SELECT m.i,
       SUM((SELECT SUM(n.j)
               FROM tab2 WHERE j=15)) AS o
    FROM tab m, tab2 n GROUP BY 1;
```

When the database server identifies the aggregate in the subquery above as a nested aggregate, it evaluates the inner **`SUM(n.j)`** aggregate in the subquery, if the outer aggregate is in the same query block where the table alias in the **`n.j`** column reference was declared.

A correlated aggregate in a subquery can include multiple correlated references to columns in the parent query block, as in the following example:

```
SELECT A.tabid,
       (SELECT SUM(B.collength * A.rowsize)
            FROM syscolumns B WHERE B.tabid = A.tabid)
    FROM systables A WHERE A.tabid = 1;
```

During execution of the statement above, the correlated aggregate

```
    SUM(B.collength * A.rowsize)
```

is evaluated in the subquery.

In all other cases, the database server treats an aggregate operating on a column of a table in a parent query block as a correlated aggregate.

Important: This behavior for evaluating correlated aggregate expressions was not fully supported in Informix 12.10.xC4 or 11.70.xC8, nor in earlier release versions.

Selecting User-Defined Function Expressions

User-defined functions extend the range of functions that are available to you and allow you to perform a subquery on each row that you select.

The following example calls the **get_orders( )** user-defined function for each **customer_num** and displays the returned value under the `n_orders` label:

```
SELECT customer_num, lname, get_orders(customer_num) n_orders
      FROM customer;
```

If an SPL routine in a SELECT statement contains certain SQL statements, the database server returns an error. For information on which SQL statements cannot be used in an SPL routine that is called within a query, see Restrictions on SPL Routines in Data-Manipulation Statements on page 1293.

For the complete syntax of user-defined function expressions, see User-Defined Functions on page 1149.

Selecting Expressions That Use Arithmetic Operators

You can combine numeric expressions with arithmetic operators to make complex expressions. You cannot combine expressions that contain aggregate functions with column expressions. These examples show expressions that use arithmetic operators within a select list in the Projection clause:

```
SELECT stock_num, quantity*total_price FROM customer;
SELECT price*2 doubleprice FROM items;
SELECT count(*)+2 FROM customer;
SELECT count(*)+LENGTH('ab') FROM customer;
```

Selecting ROW Fields

You can select a specific field of a named or unnamed ROW type column with *row.field* notation, using a period (**.**) as a separator between the *row* and *field* names. For example, suppose you have the following table structure:

```
CREATE ROW TYPE one (a INTEGER, b FLOAT);
CREATE ROW TYPE two (c one, d CHAR(10));
CREATE ROW TYPE three (e CHAR(10), f two);

CREATE TABLE new_tab OF TYPE two;
CREATE TABLE three_tab OF TYPE three;
```

The following examples show expressions that are valid in the select list:

```
SELECT t.c FROM new_tab t;
SELECT f.c.a FROM three_tab;
SELECT f.d FROM three_tab;
```

You can also enter an asterisk (*) in place of a field name to signify that all fields of the ROW-type column are to be selected.

For example, if the **my_tab** table has a ROW-type column named **rowcol** that contains four fields, the following SELECT statement retrieves all four fields of the **rowcol** column:

```
SELECT rowcol.* FROM my_tab;
```

You can also retrieve all fields from a row-type column by specifying only the column name. This example has the same effect as the previous query:

```
SELECT rowcol FROM my_tab;
```

You can use *row.field* notation not only with ROW-type columns but with expressions that evaluate to ROW-type values. For more information, see Column Expressions on page 1025 in the Expression segment.

Declaring a Display Label

You can declare a display label for any column or column expression in the select list of the Projection clause. This temporary name is in scope only while the SELECT statement is executing.

In , a display label appears as the heading for that column in the output of the SELECT statement.

In , the value of *display_label* is stored in the **sqlname** field of the **sqlda** structure. For more information on the **sqlda** structure, see the *HCL® Informix® Enterprise Replication Guide*.

If your display label is an SQL keyword, use the AS keyword to clarify the syntax. For example, to use UNITS, YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, or FRACTION as display labels, use the AS keyword with the display label. The following statement uses AS with **minute** as a display label:

```
SELECT call_dtime AS minute FROM cust_calls;
```

For the keywords of SQL, see Keywords of SQL for HCL Informix on page 1295.

If you use the INTO Table clause to create a temporary or permanent table to store the query results, you must declare a display label for any database object or expression in the select list that is not a simple column expression. The display label is used as the name of the column in the temporary or permanent table.

If you are using the SELECT statement to define a view, do not use display labels. Specify the desired label names in the CREATE VIEW column list instead.

Declaring a Column Alias

You can declare an alias for any column in the select list of the Projection clause. The GROUP BY clause can reference the column by its alias. This temporary name is in scope only while the SELECT statement is executing.

If your alias is an SQL keyword of the SELECT statement, use the AS *column_alias* keyword to clarify the syntax. For example, to use FROM as a table alias, the AS keyword must immediately precede the alias to avoid a syntax error. The following statement uses AS with **from** as an alias:

```
SELECT status AS from FROM stock GROUP BY from;
```

The following equivalent queries declare **pcol** as an alias, and use that alias in the GROUP BY clause:

```
SELECT pseudo_corinthian AS pcol FROM architecture GROUP BY pcol;
SELECTt pseudo_corinthian pcol FROM architecture GROUP BY pco1;
```

INTO Clause

Use the INTO clause in SPL routines or programs to specify the program variables or host variables to receive data that the SELECT statement retrieves.

This syntax fragment is part of the SELECT statement on page 690.

(explicit id unique_986_Connect_42_icl002) unique_986_Connect_42_icl002(explicit id unique_986_Connect_42_icl003) unique_986_Connect_42_icl003

INTO Clause

" **INTO** "

" { | *output_var* [{ { [] | : *indicator_var* | [] $ *indicator_var* | **INDICATOR** *indicator_var* } }] | *data_structure* } "

Element	Description	Restrictions	Syntax
data_ structure	Structure that was declared as a host variable	Data types of elements must be able to store the values that are being selected	Language specific
indicator_ var	Program variable to receive a return code if corresponding *output_var* receives a NULL value	Optional; use an indicator variable if the possibility exists that the value of the corresponding *output_var* is NULL	Language specific
output_ var	Program or host variable to receive value of the corresponding select list item. Can be a collection variable	Order of receiving variables must match the order of corresponding items in the select list of Projection clause	Language specific

The INTO clause specifies one or more variables that receive the values that the query returns. If it returns multiple values, they are assigned to the list of variables in the order in which you specify the variables. If the SELECT statement stands alone (that is, it is not part of a DECLARE statement and does not use the INTO clause), it must be a singleton SELECT statement. A *singleton* SELECT statement returns only one row. The following example shows a singleton SELECT statement in :

```
EXEC SQL SELECT fname, lname, company
   INTO :p_fname, :p_lname, :p_coname
   FROM customer WHERE customer_num = 101;
```

In an SPL routine, if a SELECT returns more than one row, you must use the FOREACH statement to access the rows individually. The INTO clause of the SELECT statement holds the fetched values. For more information, see FOREACH on page 935.

The number of receiving variables must be equal to the number of items in the select list of the Projection clause. The data type of each receiving variable should be compatible with the data type of the corresponding column or expression in the select list. If the data type of the receiving variable does not match that of the selected item, the data type of the selected item is converted, if possible, to the data type of the variable. If the conversion is impossible, an error occurs, and a negative value is returned in the **status** variable, **sqlca.sqlcode,** or **SQLCODE**. In this case, the value in the program variable is unpredictable. In an ANSI-compliant database, if the number of variables that are listed in the INTO clause differs from the number of items in the select list of the Projection clause, you receive an error.

In , if the number of variables listed in the INTO clause differs from the number of items in the Projection clause, a warning is returned in the **sqlwarn** structure: **sqlca.sqlwarn.sqlwarn3**. The actual number of variables that are transferred is the lesser of the two numbers. For information about the **sqlwarn** structure, see the *HCL® Informix® Enterprise Replication Guide*.

INTO Clause with Cursors

If the SELECT statement returns more than one row, you must use a cursor in a FETCH statement to fetch the rows individually. You can put the INTO clause in the FETCH statement rather than in the SELECT statement, but you should not put it in both.

The following code examples show different ways you can use the INTO clause. As both examples show, first you must use the DECLARE statement to declare a cursor.

The following example show how to use the INTO clause in the SELECT statement:

```
EXEC SQL declare q_curs cursor for
   select lname, company
      into :p_lname, :p_company
      from customer;
EXEC SQL open q_curs;
while (SQLCODE == 0)
   EXEC SQL fetch q_curs;
EXEC SQL close q_curs;
```

The following statement shows how to use the INTO clause in a FETCH statement:

```
EXEC SQL declare q_curs cursor for
   select lname, company from customer;
EXEC SQL open q_curs;
while (SQLCODE == 0)
   EXEC SQL fetch q_curs into :p_lname, :p_company;
EXEC SQL close q_curs;
```

Preparing a SELECT ... INTO Query

In , you cannot prepare a query that has an INTO clause. You can prepare the query without the INTO clause, declare a cursor for the prepared query, open the cursor, and then use the FETCH statement with an INTO clause to fetch the cursor into the program variable.

Alternatively, you can declare a cursor for the query without first preparing the query and include the INTO clause in the query when you declare the cursor. Then open the cursor and fetch the cursor without using the INTO clause of the FETCH statement.

Using Array Variables with the INTO Clause

In , if you use a DECLARE statement with a SELECT statement that contains an INTO clause, and the variable is an array element, you can identify individual elements of the array with integer literals or variables. The value of the variable that is used as a subscript is determined when the cursor is declared; the subscript variable subsequently acts as a constant.

The following code example declares a cursor for a SELECT ... INTO statement using the variables **i** and **j** as subscripts for the array **a**. After you declare the cursor, the INTO clause of the SELECT statement is equivalent to `INTO a[5], a[2]`.

```
i = 5
j = 2
EXEC SQL declare c cursor for
   select order_num, po_num into :a[i], :a[j] from orders
      where order_num =1005 and po_num =2865;
```

You can also use program variables in the FETCH statement to specify an element of a program array in the INTO clause. The program variables are evaluated at each fetch, rather than when you declare the cursor.

FROM Clause

The FROM clause of the SELECT statement lists table objects from which to retrieve data.

This syntax fragment is part of the SELECT statement on page 690.

(explicit id unique_987_Connect_42_frocl002) unique_987_Connect_42_frocl002(explicit id unique_987_Connect_42_frocl003)

unique_987_Connect_42_frocl003(explicit id unique_987_Connect_42_frocl004) unique_987_Connect_42_frocl004(explicit id unique_987_Connect_42_frocl005)

unique_987_Connect_42_frocl005(explicit id unique_987_Connect_42_frocl007) unique_987_Connect_42_frocl007(explicit id unique_987_Connect_42_frocl008)

unique_987_Connect_42_frocl008(explicit id unique_987_Connect_42_frocl010) unique_987_Connect_42_frocl010(explicit id unique_987_Connect_42_tabx003)

unique_987_Connect_42_tabx003(explicit id unique_987_Connect_42_tabx001) unique_987_Connect_42_tabx001(explicit id unique_987_Connect_42_frocl011)

unique_987_Connect_42_frocl011(explicit id unique_987_Connect_42_frocl012) unique_987_Connect_42_frocl012

FROM Clause

" **FROM** "

" { | (<ANSI Tables> { <ANSI Joins> [] }) | { <ANSI Tables> <Other Tables> } [[] , <Informix OUTER Join Clause> []] | [<Lateral derived table> []] } "

ANSI Tables

" { { { *synonym* | *table* | *view* | **ONLY** { *(table)* | *(synonym)* } } } [[**AS**] *alias* []] | { [] | <Collection Derived Table> [] | <Iterator> [] } } "

Other Tables

" { { *external* } [[**AS**] *alias* [] [(*derived_column*)]] | ([] <Collection Subquery>) [] | (*subquery*) | { { *table* | *view* | *synonym* } | [] **ONLY** ([] { *table* | *synonym* }) [] } [**AS**] *alias* [] (*derived_column*) } "

Element	Description	Restrictions	Syntax
alias	Temporary name for a table, view, or derived table in this query	See Aliases for Tables or Views on page 713.	Identifier on page 1231
derived_column	Temporary name for a derived column in a table expression	Unless the underlying collection is a ROW type, you can declare no more than one *derived_column* name	Identifier on page 1231
external	External table from which to retrieve data	Must exist, but cannot be the outer table in an outer join	Database Object Name on page 1225
subquery	Nested query whose results are available to the outer query	See Table expressions on page 714	SELECT statement on page 690

Element	Description	Restrictions	Syntax
synonym, table, view	Synonym for a table from which to retrieve data	Synonym and table or view to which it points must exist. Must include appropriate qualifiers.	Database Object Name on page 1225

Every SELECT statement requires the FROM clause, whether or not any data source is required. If your query uses the database server to evaluate an expression that requires no data source, the FROM clause can reference any existing table in the current database on which you hold sufficient access privileges, as in the following example:

```
SELECT ATANH(SQRT(POW(4,2) + POW(5,2))) FROM systables;
```

Aliases for Tables or Views

You can declare an alias for a table or view in the FROM clause. If you do so, you must use the alias to refer to the table or view in other clauses of the SELECT statement. You can also use aliases to make the query shorter.

The following examples show typical uses of the FROM clause. The first query selects all the columns and rows from the **customer** table. The second query uses a join between the **customer** and **orders** table to select all the customers who have placed orders.

```
SELECT * FROM customer;
SELECT fname, lname, order_num FROM customer, orders
   WHERE customer.customer_num = orders.customer_num;
```

The next example is equivalent to the second query in the preceding example, but it declares aliases in the FROM clause and uses them in the WHERE clause:

```
SELECT fname, lname, order_num FROM customer c, orders o
   WHERE c.customer_num = o.customer_num;
```

Aliases (sometimes called *correlation names*) are especially useful with a self-join. For more information about self-joins, see Specifying a Join in the WHERE Clause on page 736. In a self-join, you must list the table name twice in the FROM clause and declare a different alias for each of the two instances of the table name.

If you use a potentially ambiguous word as an alias (or as a display label), you must begin its declaration with the keyword AS. This keyword is required if you use any of the keywords ORDER, FOR, AT, GROUP, HAVING, INTO, NOT, UNION, WHERE, WITH, CREATE, or GRANT as an alias for a table or view.

The database server would issue an error if the next example did not include the AS keyword to indicate that **not** is a display label, rather than an operator:

```
CREATE TABLE t1(a INT);
SELECT a AS not FROM t1;
```

If you do not declare an alias for a collection-derived table, the database server assigns an implementation-dependent name to it.

Table expressions

A *table expression* (sometimes called a *derived table*) is the name of a table or view, or a specification that evaluates to a set of rows. These rows are typically the result of a query that is embedded in a nested SELECT statement, or in some other SQL statement.

Table expressions can have the following syntax. These options are a subset of the table options in the ANSI Table Reference and the Other Table Reference in the FROM Clause on page 711.

Table expression

(explicit id unique_997_Connect_42_tabx001) unique_997_Connect_42_tabx001(explicit id unique_997_Connect_42_tabx002)
unique_997_Connect_42_tabx002(explicit id unique_997_Connect_42_tabx003) unique_997_Connect_42_tabx003(explicit id unique_997_Connect_42_tabx004)
unique_997_Connect_42_tabx004(explicit id unique_997_Connect_42_tabx005) unique_997_Connect_42_tabx005 { *table_object* | (*subquery*) | [`<Collection-Derived Table>`] | ([`<Collection Subquery>`)] | `<Lateral derived table>` [| [`<Iterator>`] }

Element	Description	Restrictions	Syntax
subquery	Nested query whose results are available to the outer query	See **Usage** notes below	SELECT statement on page 690
table _object	Name, synonym, or alias of a table, view, or EXTERNAL table	Must exist, or must reference a derived table that the SELECT statement creates	Identifier on page 1231 or Database Object Name on page 1225

Usage

Table expressions can be simple or complex:

- Simple table expressions

  A *simple* table expression is one whose underlying query can be folded into the main query while preserving the correctness of the query result.

- Complex table expressions

  A *complex* table expression is one whose underlying query cannot be folded into the main query while preserving the correctness of the query result. The database server materializes such table expressions into a temporary table that is used in the main query. Subqueries in the FROM clause that specify aggregates, set operators, or the ORDER BY clause are implemented as complex table expressions, which typically require more resources of the database server than simple table expressions.

In either case, the table expression is evaluated as a general SQL query and its results can be thought of as a logical table. This logical table and its columns can be used just like an ordinary base table, but it is not persistent. It exists only during the execution of the query that references it.

Restrictions on table expressions

Table expressions have the same syntax as general SELECT statements, but with most of the restrictions that apply to subqueries in other contexts. A table expression cannot include the SELECT INTO clause that explicitly creates a result table.

Informix® does not support Generalized Key indexes. It supports table expressions in the triggered actions of CREATE TRIGGER statements, and as the triggering event of a Select trigger. Informix® also supports the ORDER BY clause in table expressions.

Informix® supports iterator functions as FROM clause table expressions. The CALL statement of SPL, however, cannot invoke an iterator TABLE function within a subquery in the FROM clause.

Apart from these restrictions, any valid SQL query can be a table expression. A table expression can be nested within another table expression and can include tables and views in its definition. You can use table expressions in CREATE VIEW statements to define views.

Correlated subqueries and derived tables

A *correlated subquery* is a subquery that refers to a column of a table that is not listed in its FROM clause. Conversely, any subquery that references only columns in tables that are listed in its FROM clause is an *uncorrelated subquery*.

In the following example, the uncorrelated subquery that defines a derived table in its FROM clause contains a correlated subquery in its WHERE clause:

```
SELECT * FROM (SELECT * FROM t1
   WHERE a IN (SELECT b FROM t2 WHERE t1.a = t2.b));
```

Here the subquery in the first WHERE clause is a correlated subquery, because it references column **a** of table **t1**, but its FROM clause specifies only table **t2**.

In FROM clause table expressions, HCL Informix® also supports the ORDER BY clause, which is not valid in subqueries outside the FROM clause. Columns or expressions that are specified by the ORDER BY clause in a table expression need not be included in the Projection clause.

Lateral derived tables

The LATERAL keyword must immediately precede any query in the FROM clause that defines a derived table, if that query references any other table or column that appears earlier in the same FROM clause than the query that defines the derived table.

Lateral derived tables, and the scope of reference of the table and column aliases that can be declared in their syntax, are part of the ISO/ANSI standard for the SQL language. This syntax fragment is part of the FROM Clause on page 711.

LATERAL derived table

" **LATERAL** (*subquery*) **[AS]** *alias* [(*column_alias*)] "

Element	Description	Restrictions	Syntax
alias	Temporary name declared here for the derived table of *subquery* results	See Aliases for Tables or Views on page 713.	Identifier on page 1231
column_alias	Temporary name declared here for a column in the derived table		Identifier on page 1231
subquery	Specifies rows to be retrieved	Can be uncorrelated or correlated	SELECT statement on page 690

Usage

The LATERAL keyword is required if the *subquery* whose result set is the derived table references any table or column that appears earlier in the same FROM clause. Here *earlier* means "to the left of the derived-table" in the left-to-right order of syntax tokens in the FROM clause. A derived table defined with the LATERAL keyword is called a *lateral derived table*.

This support for references to columns in other tables in the FROM clause, rather than only to columns in subsequent derived tables, can improve performance in SELECT statements that join one or more derived tables. Lateral table and column references are also valid in derived tables within DELETE, UPDATE, and CREATE VIEW statements.

The LATERAL keyword is not required in the FROM clause for derived tables in which all uncorrelated table and column references have already been resolved.

Restrictions on lateral correlated references

The following restrictions apply to lateral derived table and column references:

- They cannot be used in ANSI FULL OUTER JOIN queries.
- They cannot be used in ANSI RIGHT OUTER JOIN queries.
- They cannot be used in Informix®-extension OUTER JOIN queries.

Examples of lateral derived tables

The following query includes a lateral derived table in the FROM clause, where t1_a is a lateral correlation reference:

```
SELECT * FROM t1 ,
            LATERAL (SELECT t2.a AS t2_a
                     FROM t2 WHERE t2.a = t1.a);
```

In the next example, d.deptno is a lateral correlation reference:

```
   SELECT d.deptno, d.deptname,
      empinfo.avgsal, empinfo.empcount
         FROM department d,
            LATERAL (SELECT AVG(e.salary) AS avgsal,
                            COUNT(*) AS empcount
                     FROM employee e
                     WHERE e.workdept=d.deptno) AS empinfo;
```

Here the avgsal and empcount aliases for column expressions and the empinfo lateral table reference appear in the projection list of the outer query, which joins qualifying rows from the **department** table and the derived table, using the correlation deptno.

Usability and Performance Considerations

Although equivalent functionality is available through views, subqueries as table expressions simplify the formulation of queries, make the syntax more flexible and intuitive, and support the ANSI/ISO standard for SQL.

The query optimizer does not materialize simple table expressions that the FROM clause specifies. The performance of a query that uses the ANSI/ISO syntax for a table expression in the FROM clause is at least as good as a that of a query that uses the Informix®-extension TABLE (MULTISET (SELECT ...)) syntax to specify an equivalent derived table in the FROM clause. Subqueries in the FROM clause that include aggregate functions, set operators like the UNION, INTERSECT, or MINUS operators, or ORDER BY specifications are implemented as complex table expressions that can impose greater costs than simple table expressions. Use the SET EXPLAIN statement to examine the query plan and the estimated cost of a table expression.

The following are examples of valid table expressions:

```
SELECT * FROM (SELECT * FROM t);

SELECT * FROM (SELECT * FROM t) AS s;

SELECT * FROM (SELECT * FROM t) AS s WHERE t.a = s.b;

SELECT * FROM (SELECT * FROM t) AS s, (SELECT * FROM u) AS v WHERE s.a = v.b;

SELECT * FROM (SELECT SKIP 2 col1 FROM tab1 WHERE col1 > 50 ORDER BY col1  DESC);

SELECT * FROM (SELECT col1,col3 FROM tab1
   WHERE col1 < 50 GROUP BY col1,col3 ORDER BY col3 ) vtab(vcol0,vcol1);

SELECT * FROM (SELECT * FROM t WHERE t.a = 1) AS s,
OUTER
(SELECT * FROM u WHERE u.b = 2 GROUP BY 1) AS v WHERE s.a = v.b;

SELECT * FROM (SELECT a AS colA FROM t WHERE t.a = 1) AS s,
OUTER
(SELECT b AS colB FROM u WHERE u.b = 2 GROUP BY 1) AS v
   WHERE s.colA = v.colB;

CREATE VIEW vu AS SELECT * FROM (SELECT * FROM t);

SELECT * FROM ((SELECT * FROM t) AS r) AS s;
```

Restrictions on External Tables in Joins and Subqueries

When you use external tables in joins or subqueries, the following restrictions apply:

- No more than one external table is valid in a query.
- The external table cannot be the outer table in an outer join.

- For subqueries that cannot be converted to joins, you can use an external table in the main query, but not in the subquery.
- You cannot do a self-join on an external table.

For more information on subqueries, see your .

The ONLY Keyword

If the FROM clause includes a permanent table that is a supertable within a typed table hierarchy, the query returns the qualifying rows from both the supertable and from its subtables by default, unless you specify the ONLY keyword.

For the SELECT statement to return rows from the supertable only, you must include the ONLY keyword immediately before the supertable name in the FROM clause, and you must enclose the identifier or synonym of the supertable within parentheses, as in this example:

```
SELECT * FROM ONLY(super_tab);
```

The data sources for this query do not include the subtables of **super_tab**.

Selecting from a Collection Variable

The SELECT statement in conjunction with the Collection-Derived Table segment allows you to select elements from a collection variable.

About this task

The Collection-Derived Table segment identifies the collection variable from which to select the elements. (See Collection-Derived Table on page 1212.)

To modify the contents of a column of a collection data type, you can use the SELECT statement with a collection variable in various ways:

- You can select the contents (if any) of a collection column into a collection variable.

  You can assign the data type of the column to a collection variable of type COLLECTION (that is, an untyped collection variable).
- You can select the contents from a collection variable to determine the data that you might want to update.
- You can select the contents from a collection variable INTO another variable in order to update certain collection elements.

  The INTO clause identifies the variable for the element value that is selected from the collection variable. The data type of the host variable in the INTO clause must be compatible with that of the corresponding collection element.
- You can use a Collection cursor to select one or more elements from the collection variable.

  For more information, including restrictions on the SELECT statement, see Associating a Cursor with a Prepared Statement on page 445.

- You can use a Collection cursor to select one or more elements from an SPL collection variable.

  For more information, including restrictions on the SELECT statement, see Using a SELECT ... INTO Statement on page 937.

When one of the tables to be joined is a collection, the FROM clause cannot specify a join. This restriction applies when the collection variable holds your collection-derived table. See also Collection-Derived Table on page 1212. and the INSERT, UPDATE, and DELETE statement descriptions in this chapter.

Selecting from a Row Variable (ESQL/C)

The SELECT statement can include the Collection-Derived Table segment to select one or more fields from a **row** variable.

About this task

The Collection-Derived Table segment identifies the **row** variable from which to select the fields. For more information, see Collection-Derived Table on page 1212.

To select fields:

1. Create a **row** variable in your program.
2. Optionally, fill the **row** variable with field values.
   You can select a ROW-type column into the **row** variable with the SELECT statement (without the Collection-Derived Table segment). Alternatively, you can insert field values into the **row** variable with the UPDATE statement and the Collection-Derived Table segment.
3. Select row fields from the **row** variable with the SELECT statement and the Collection-Derived Table segment.
4. Once the **row** variable contains the correct field values, you can use the INSERT or UPDATE statement on a table or view name to save the contents of the **row** variable in a named or unnamed row column.

Results

The INTO clause can specify a host variable to hold a field value selected from the **row** variable.

The type of the host variable must be compatible with that of the field. For example, this code fragment puts the **width** field value into the **rect_width** host variable.

```
EXEC SQL BEGIN DECLARE SECTION;
   ROW (x INT, y INT, length FLOAT, width FLOAT) myrect;
   double rect_width;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL SELECT rect INTO :myrect FROM rectangles
   WHERE area = 200;
EXEC SQL SELECT width INTO :rect_width FROM table(:myrect);
```

The SELECT statement on a **row** variable has the following restrictions:

- No expressions are allowed in the select list of the Projection clause.
- ROW columns cannot be in a WHERE clause comparison condition.

- The Projection clause must be an asterisk (*) if the row-type contains fields of opaque, distinct, or built-in data types.
- Columns listed in the Projection clause can have only unqualified names. They cannot use the syntax *database@server:table.column.*
- The following clauses are not allowed: GROUP BY, HAVING, INTO TEMP, ORDER BY, and WHERE.
- The FROM clause has no provisions to do a join.

You can modify the **row** variable with the Collection-Derived Table segment of the UPDATE statements. (The INSERT and DELETE statements do not support a **row** variable in the Collection-Derived Table segment.)

The **row** variable stores the fields of the row. It has no intrinsic connection, however, with a database column. Once the **row** variable contains the correct field values, you must then save the variable into the ROW column with one of the following SQL statements:

- To update the ROW column in the table with the **row** variable, use an UPDATE statement on a table or view name and specify the **row** variable in the SET clause. For more information, see Updating ROW-Type Columns on page 876.
- To insert a row into a ROW column, use the INSERT statement on a table or view and specify the **row** variable in the VALUES clause. See Inserting Values into ROW-Type Columns on page 586.

For examples of how to use SPL row variables, see the *HCL® Informix® Guide to SQL: Tutorial*. For information on using **row** variables, see the discussion of complex data types in the *HCL® Informix® Enterprise Replication Guide*.

Iterator Functions

The FROM clause can include a call to an iterator function to specify the source for a query. An *iterator function* is a user-defined function that returns to its calling SQL statement multiple times, each time returning at least one value.

You can query the returned result set of an iterator UDR using a virtual table interface. This syntax fragment is part of the FROM Clause on page 711.

(explicit id unique_1000_Connect_42_itfu003) unique_1000_Connect_42_itfu003(explicit id unique_1000_Connect_42_itfu002) unique_1000_Connect_42_itfu002

Iterator

" TABLE ([[] { FUNCTION | PROCEDURE }] "

" *iterator* ([<Routine Parameter List> []])) "

" [{ [AS] *table* [(*column*)] }] "

Element	Description	Restrictions	Syntax
column	Name declared here for a virtual column in *table*	Must be unique among *column* names in *table*, and cannot include qualifiers.	Identifier on page 1231

Element	Description	Restrictions	Syntax
iterator	Name of the iterator function	Must be registered in the database	Identifier on page 1231
table	Name declared here for virtual table holding the *iterator* result set	Cannot include qualifiers	Identifier on page 1231

The keyword FUNCTION (or PROCEDURE) was required in releases earlier than Informix® 10.5. These keyword extensions to the ANSI/ISO standard for SQL are optional in this release, and have no effect. The following two query specifications, which specify **fibGen( )** as an iterator function, are equivalent:

```
SELECT * FROM TABLE FUNCTION ( fibGen(10));
SELECT * FROM TABLE ( fibGen(10));
```

The *table* can only be referenced within the context of this query. After the SELECT statement terminates, the virtual table no longer exists.

The number of columns must match the number of values returned by the iterator. An external function can return no more than one value (but that can be of a collection data type). An SPL routine can return multiple values.

The database server issues error -595, however, if any argument to the iterator *table* function is an aggregate expression.

To reference the virtual *table* columns in other parts of the SELECT statement, for example, in the WHERE clause or HAVING clause, you must declare its name and the virtual column names in the FROM clause. You do not need to declare the *table* name or *column* names in the FROM clause if you use the asterisk notation in the Select list of the Projection clause:

```
SELECT * FROM ...
```

For more information and examples of using iterator functions in queries, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

ANSI Joins

If the FROM clause specifies more than one table reference, the query can join rows from several tables or views. A *join condition* specifies a relationship between at least one column from each table to be joined. Because the columns in a join condition are being compared, they must have compatible data types.

This syntax fragment is part of the FROM Clause on page 711.

(explicit id unique_1001_Connect_42_ajt003) unique_1001_Connect_42_ajt003(explicit id unique_1001_Connect_42_ajt004)

unique_1001_Connect_42_ajt004(explicit id unique_1001_Connect_42_ajt006) unique_1001_Connect_42_ajt006(explicit id unique_1001_Connect_42_ajt007)

unique_1001_Connect_42_ajt007(explicit id unique_1001_Connect_42_ajt008) unique_1001_Connect_42_ajt008(explicit id unique_1001_Connect_42_specj002)

unique_1001_Connect_42_specj002(explicit id unique_1001_Connect_42_ajt009) unique_1001_Connect_42_ajt009

ANSI Joins

" { { **INNER** | [{ **LEFT** | **RIGHT** | **FULL** } [**OUTER**]] } **JOIN** <ANSI Tables> [] <ON Clause> | [**CROSS JOIN** <ANSI Tables> []] } "

" [[**AS**] *alias* []] "

ON Clause

" **ON** "

" { | { | [{ *alias* . | *external* . | *table* . | *view* . | *synonym* . }] *column* <Relational Operator> [] [{ *alias* . | *external* . | *table* . | *view* . | *synonym* . }] *column* | <Function Expression> [] | <Condition> [] | (*subquery*) | (<Collection Subquery> []) } } "

[]

Element	Description	Restrictions	Syntax
alias	Temporary alternative name declared in the FROM clause for a table or view	See Aliases for Tables or Views on page 713	Identifier on page 1231
column	Column of a table or view to be joined	Must exist in the table or view	Identifier on page 1231
synonym, *table*, *view*	Name of a synonym, table, or view to be joined in the query	Synonym and the table or view to which it points must exist. Must include appropriate qualifiers.	Database Object Name on page 1225
subquery	Embedded query	Cannot contain the FIRST or the ORDER BY clause	SELECT statement on page 690

You must use the same form of join syntax (either Informix®® extension or ANSI-compliant) for all of the outer joins in the same query block. When you use the ANSI-compliant join syntax, you must also specify the join condition in the ON clause.

Rows from the tables or views are *joined* when there is a match between the values of specified columns. When the columns to be joined have the same name, you must qualify each column name with its data source. If you join a table to itself, you must list the table name twice in the FROM clause and assign it two different table aliases. Then use the aliases to qualify the column names.

By default, the order in which the database server joins tables and views is independent of the order in which they are referenced in the FROM clause. To force the order in which table objects are joined to match the FROM clause order, you can specify the ORDERED optimizer directive after the SELECT keyword. For more information, see the section Join-Order Directive on page 1254.

Join-method optimizer directives that you specify for an ANSI-compliant joined query are ignored, but are listed under *Directives Not Followed* in the explain output file.

Multiple join specifications require parentheses around the original table and join segment

The ANSI-Joined Table segment must be enclosed between parentheses if it is immediately followed by another join specification. For example, the first of the following two queries returns an error; the second query is valid:

```
SELECT * FROM (T1 LEFT JOIN T2) CROSS JOIN T3 ON (T1.c1 = T2.c5)
      WHERE (T1.c1 < 100);    -- Ambiguous order of operations;

SELECT * FROM (T1 LEFT JOIN T2 ON (T1.c1 = T2.c5)) CROSS JOIN T3
      WHERE (T1.c1 < 100);    -- Unambiguous order of operations;
```

The following valid query specifies nested LEFT OUTER joins of table expressions within the FROM clause of the outer SELECT statement:

```
SELECT * FROM
( (SELECT C1,C2 FROM T3) AS VT3(V31,V32)
 LEFT OUTER JOIN
        ( (SELECT C1,C2 FROM T1) AS VT1(VC1,VC2)
        LEFT OUTER JOIN
        (SELECT C1,C2 FROM T2) AS VT2(VC3,VC4)
        ON VT1.VC1 = VT2.VC3)
ON VT3.V31 = VT2.VC3);
```

Join keyword definitions

In an ANSI-compliant join that specifies the LEFT, RIGHT, or FULL keywords in the FROM clause, the OUTER keyword is optional.

Table 12. Join keywords

FROM Clause Keywords	Corresponding Result Set
CROSS JOIN	The Cartesian product, which is all possible paired combinations that include one row from each of the joined tables.

Table 12. Join keywords (continued)

FROM Clause Keywords	Corresponding Result Set
INNER JOIN	All the rows in a table that have one or more matching rows in the other table (or tables). The unmatched rows are discarded.
LEFT OUTER JOIN	All the rows of the first table reference and the qualifying rows from the second table reference The LEFT keyword specifies a join that treats the first table reference as the dominant table in the join. In a left outer join, the subordinate part of the outer join appears to the right of the keyword that begins the outer join specification. The result set includes all the rows that an INNER join returns, plus all rows that would otherwise have been discarded from the dominant table.
RIGHT OUTER JOIN	The qualifying rows from the first table reference and all the rows of the second table reference The RIGHT keyword specifies a join that treats the second table reference as the dominant table in the join. In a right outer join, the subordinate part of the outer join appears to the left of the keyword that begins the outer join specification. The result set includes all the rows that an INNER join returns, plus all rows that would otherwise have been discarded from the dominant table.
FULL OUTER JOIN	The union of all rows from an INNER join of the two tables, and of all rows of each table that have no match in the other table (using NULL values in the selected columns of the other table)

Using the ON Clause

Use the ON clause to specify the join condition and any expressions as optional join filters.

The following example from the **stores_demo** database illustrates how the join condition in the ON clause combines the **customer** and **orders** tables:

```
SELECT c.customer_num, c.company, c.phone, o.order_date
   FROM customer c LEFT JOIN orders o
      ON c.customer_num = o.customer_num;
```

The following table shows part of the joined **customer** and **orders** tables.

customer_num	company	phone	order_date
101	All Sports Supplies	408-789-8075	05/21/2008

customer_num	company	phone	order_date
102	Sports Spot	415-822-1289	NULL
103	Phil's Sports	415-328-4543	NULL
104	Play Ball!	415-368-1100	05/20/2008
—	—	—	—

In an outer join, the join filters (expressions) that you specify in the ON clause determine which rows of the subordinate table join to the dominant (or outer) table. The dominant table, by definition, returns all its rows in the joined table. That is, a join filter in the ON clause has no effect on the dominant table.

If the ON clause specifies a join filter on the dominant table, the database server joins only those dominant table rows that meet the criterion of the join filter to rows in the subordinate table. The joined result contains all rows from the dominant table. Rows in the dominant table that do not meet the criterion of the join filter are extended with NULL values for the subordinate columns.

The following example from the **stores_demo** database illustrates the effect of a join filter in the ON clause:

```
SELECT c.customer_num, c.company, c.phone, o.order_date
   FROM customer c LEFT JOIN orders o
      ON c.customer_num = o.customer_num
         AND c.company <> "All Sports Supplies";
```

The row that contains All Sports Supplies remains in the joined result.

customer_num	company	phone	order_date
101	All Sports Supplies	408-789-8075	NULL
102	Sports Spot	415-822-1289	NULL
103	Phil's Sports	415-328-4543	NULL
104	Play Ball!	415-368-1100	05/20/2008
—	—	—	—

Even though the order date for customer number 101 is 05/21/2008 in the **orders** table, the effect of placing the join filter (`c.company <> "All Sports Supplies"`) prevents this row in the dominant **customer** table from being joined to the subordinate **orders** table. Instead, a NULL value for **order_date** is extended to the row of All Sports Supplies.

Applying a join filter to a base table in the subordinate part of an outer join can improve performance. For more information, see your .

Specifying a Post-Join Filter

When you use the ON clause to specify the join, you can use the WHERE clause as a post-join filter. The database server applies the post-join filter of the WHERE clause to the results of the outer join.

The following example illustrates the use of a post-join filter. This query returns data from the **stores_demo** database. Suppose you want to determine which items in the catalog are not being ordered. The next query creates an outer join of the data from the **catalog** and **items** tables and then determines which catalog items from a specific manufacturer (HRO) have not sold:

```
SELECT c.catalog_num, c.stock_num, c.manu_code, i.quantity
   FROM catalog c LEFT JOIN items i
   ON c.stock_num = i.stock_num AND c.manu_code = i.manu_code
   WHERE i.quantity IS NULL AND c.manu_code = "HRO";
```

The WHERE clause contains the post-join filter that locates the rows of HRO items in the catalog for which nothing has been sold.

When you apply a post-join filter to a base table in the dominant or subordinate part of an outer join, you might improve performance. For more information, see your .

Using a Join as the Dominant or Subordinate Part of an Outer Join

With the ANSI join syntax, you can nest joins. You can use a join as the dominant or subordinate part of an outer or inner join.

Suppose you want to modify the previous query (the post-join filter example) to get more information that will help you determine whether to continue carrying each unsold item in the catalog. You can modify the query to include information from the **stock** table so that you can see a short description of each unsold item with its cost:

```
SELECT c.catalog_num, c.stock_num, s.description, s.unit_price,
   s.unit_descr, c.manu_code, i.quantity
FROM (catalog c INNER JOIN stock s
   ON c.stock_num = s.stock_num
      AND c.manu_code = s.manu_code)
    LEFT JOIN items i
      ON c.stock_num = i.stock_num
         AND c.manu_code = i.manu_code
   WHERE i.quantity IS NULL
      AND c.manu_code = "HRO";
```

In this example, an inner join between the **catalog** and **stock** tables forms the dominant part of an outer join with the **items** table.

For additional examples of outer joins, see the *HCL® Informix® Guide to SQL: Tutorial*.

Informix®-Extension Outer Joins

The Informix®-extension syntax for outer joins begins with an implicit left outer join. That is, you begin the Informix®-extension outer join with the OUTER keyword.

This syntax fragment is part of the FROM Clause on page 711.

(explicit id unique_1002_Connect_42_ieoj002) unique_1002_Connect_42_ieoj002(explicit id unique_1002_Connect_42_ieej003)

unique_1002_Connect_42_ieej003

Informix OUTER Clause

"OUTER"

"{{ <ANSI Tables> [] | <Other Tables> [] } | ({| { <ANSI Tables> [] | <Other Tables> [] } , [] <Informix OUTER Clause> | [] <Informix OUTER Clause> , { <ANSI Tables> [] | <Other Tables> [] } }) }"

The following example uses the OUTER keyword to create an outer join that lists all customers and their orders, regardless of whether they have placed orders:

```
SELECT c.customer_num, c.lname, o.order_num FROM customer c,
   OUTER orders o WHERE c.customer_num = o.customer_num;
```

This example returns all the rows from the **customer** table with the rows that match in the **orders** table. If no record for a customer appears in the **orders** table, the returned **order_num** column for that customer has a NULL value.

If you have a complex outer join, that is, the query has more than one outer join, you must either embed the additional outer join or joins in parentheses, as the syntax diagram shows, or establish join conditions, or relationships, between the dominant table and each subordinate table in the WHERE clause.

When an expression or a condition in the WHERE clause relates two subordinate tables, you must use parentheses around the joined tables in the FROM clause to enforce dominant-subordinate relationships, as in this example:

```
SELECT c.company, o.order_date, i.total_price, m.manu_name
   FROM customer c,
      OUTER (orders o, OUTER (items i, OUTER manufact m))
   WHERE c.customer_num = o.customer_num
      AND o.order_num = i.order_num
      AND i.manu_code = m.manu_code;
```

When you omit parentheses around the subordinate tables in the FROM clause, you must establish join conditions between the dominant table and each subordinate table in the WHERE clause. If a join condition is between two subordinate tables, the query fails.

The following example, however, successfully returns a result

- that joins the dominant **customer** table with the subordinate **orders** table,
- and joins the dominant **customer** table with the subordinate **cust_calls** table:

```
SELECT c.company, o.order_date, c2.call_descr
   FROM customer c, OUTER orders o, OUTER cust_calls c2
   WHERE c.customer_num = o.customer_num
      AND c.customer_num = c2.customer_num;
```

The *HCL® Informix® Guide to SQL: Tutorial* has examples of complex outer joins.

Restrictions on Informix®-extension outer joins

If you use this Informix®-extension syntax for an outer join, all of the following restrictions apply to the same SELECT statement:

- You must use Informix®-extension syntax for all outer joins in a single query block.
- You must include the join condition in the WHERE clause.
- You cannot begin another outer join with the LEFT JOIN or the LEFT OUTER JOIN keywords.
- You cannot define a lateral table reference or include the LATERAL keyword.
- Within the Informix®-extension outer join, the Table Reference syntax segment cannot include a lateral table reference that is declared in the same SELECT statement.

GRID clause

Use the GRID clause to specify the scope of a cross-server query whose data sources are tables of database servers that are nodes of a grid.

The GRID clause is not valid unless the session is connected to a database within an existing grid. A grid can be created by using appropriate cdr commands and ifx_grid routines of the Enterprise Replication facility.

This syntax is an extension to the ANSI/ISO standard for the SQL language.

This syntax fragment is part of the SELECT statement on page 690.

GRID clause



Element	Description	Restrictions	Syntax
grid	Name of the grid that is in scope for this query	Must exist and must be defined by the cdr define grid command	Identifier on page 1231
region	Name of the region that is in scope for this query	Must exist and must be defined by the cdr define region command	Identifier on page 1231

Usage

Any SELECT statement that explicitly or implicitly includes the GRID clause is called a *grid query*. The result of a grid query are qualifying rows from a logical UNION or UNION ALL of each table in the FROM clause across tables with the same names and the same schema in every grid server. This union can include tables across all nodes in the grid, or across a subset of those grid nodes, called a *region*.

Note: The tables that the FROM clause of a grid query specifies must all have the same schema and must meet other requirements that this topic identifies. Because of these restrictions, not all Informix® grids can support grid queries.

The optional ALL keyword

If the optional ALL keyword immediately follows the GRID keyword, the result of the grid query is a logical UNION ALL. meaning that the result set of the grid query can include duplicate rows. Otherwise, if you omit the ALL keyword, only distinct values are returned from the logical UNION of the results from each participating grid server.

SET ENVIRONMENT statement options for grid queries

Two options to the SET ENVIRONMENT statement can define a default GRID clause, so that any subsequent SELECT statements with no GRID clause is interpreted as grid query that includes the default GRID clause:

SET ENVIRONMENT SELECT_GRID

This statement can specify a grid or region as the default scope of subsequent grid queries that return the union of unique qualifying rows. The GRID clause can omit the grid or region name for grid queries that return UNION results for the specified default nodes.

SET ENVIRONMENT SELECT_GRID_ALL

This statement can specify a grid or region as the default scope of subsequent grid queries that return the union of all qualifying rows, including duplicates. The GRID clause can omit the grid or region name for grid queries that return UNION ALL results for the specified default nodes.

While either of these options to the SET ENVIRONMENT statement is enabled, the SQL parser applies the current default GRID clause to every SELECT statement in the session that does not include an explicit GRID clause. No more than one default GRID clause can be in effect during the same session at the same point in time. When either option is in effect, using the SET ENVIRONMENT statement to set the other keyword option, or to reset the same keyword option for a different grid or region disables the previously set default.

You can also disable the default GRID clause by issuing either of these SQL statements:

```
SET ENVIRONMENT SELECT_GRID DEFAULT;_
SET ENVIRONMENT SELECT_GRID_ALL DEFAULT;
```

Each of the statements above prevents the database server from interpreting every subsequent query in the current session as a grid query. Unless you define a new default GRID clause in the same session, any subsequent SELECT statement must include an explicit GRID clause to run as a grid query.

While either the SELECT_GRID session environment option for UNION queries or the SELECT_GRID_ALL session environment option for UNION ALL queries has specified a default grid or region as the scope of grid queries in the current session, you can omit the GRID clause in grid queries on the nodes of that grid or region, as in the following example, where **tab1** and **tab2** are tables that have the same schema, locale, and code set on every grid server within the **region_03** subset of a grid:

```
SET ENVIRONMENT SELECT_GRID 'region_03'
SELECT * FROM tab1;
SELECT * FROM tab2;
```

The two queries above are executed as if you had explicitly specified this GRID clause:

```
SELECT * FROM tab1 GRID 'region_03';
SELECT * FROM tab2 GRID 'region_03';
```

No more than one of the SELECT_GRID and SELECT_GRID_ALL session environment options to the SET ENVIRONMENT statement can be enabled for the current user session at the same point in time. When either option is in effect, using the SET ENVIRONMENT statement to set the other keyword option, or to reset the same keyword option for a different grid or region disables the previously set default.

The following SQL statements replace the previous default GRID clause by defining a different default GRID clause that combines the UNION ALL results from participating grid servers in a different region, the **region_04** subset of a grid:

```
SET ENVIRONMENT SELECT_GRID_ALL 'region_04'
SELECT * FROM tab1;
SELECT * FROM tab2;
```

Those two queries will be executed as if you had specified this GRID clause:

```
SELECT * FROM tab1 GRID ALL 'region_04';
SELECT * FROM tab2 GRID ALL 'region_04';
```

By default, a grid query fails if the database server that issues the grid query cannot connect to one or more of the nodes within the grid or region that the explicit or default GRID clause specifies. Another session environment variable that the SET ENVIRONMENT statement can enable can return partial results from a grid query, even if some grid servers in the specified grid or region are unavailable:

SET ENVIRONMENT GRID_NODE_SKIP

This statement can enable processing of a grid query to continue when one or more of the grid servers is unavailable.

If you issue the SQL statement

```
SET ENVIRONMENT GRID_NODE_SKIP ON;
```

the database server ignores any node which is not available, and returns qualifying rows from the participating grid servers. You can identify any skipped nodes by invoking the ifx_gridquery_skipped_nodes() function.

Another function, ifx_gridquery_skipped_node_count(), can be used to detect how many nodes were skipped. For more information about these functions, see the .

Tables in the FROM clause of grid queries

Only permanent database tables are valid in the FROM clause of a grid query. They must be defined as grid tables by running the `cdr change gridtable` command.

The following table objects are not supported:

- Synonyms or views on tables, except for tables in the **sysmaster** database
- Table objects that the CREATE EXTERNAL TABLE statement defined
- Tables that are qualified by the name of a database server or grid server
- Tables on which a concurrent ALTER TABLE, ALTER FRAGMENT, or ALTER INDEX operation is being performed
- Tables that have a different schema from other tables of the same name in databases of participating grid servers
- Tables in databases that were not created with the same database locale and code set
- Tables in databases whose settings for the SQL_LOGICAL_CHAR configuration parameter or the **DELIMIDENT** or **GL_USEGLU** environment variables are not the same across all databases participating in the grid query.

In addition, the projection list of a grid query cannot include any column or expression whose data type is not supported in cross-server queries. The unsupported data types include all complex or large-object types, and some user-defined types (UDTs) and opaque types.

The same restrictions that apply to DISTINCT data types in distributed DML operations across databases of the same Informix® instance also apply to DISTINCT data types in grid queries. For a discussion of the data types that are valid in distributed queries, see the topics Data Types in Cross-Server Transactions on page 702 and DISTINCT Types in Distributed Operations on page 996.

Additional restrictions on grid queries

The user executing the grid query must be a valid user on all nodes within the grid or region.

A grid query cannot be a subquery that contains references to its outer query.

The grid query cannot reference any of the following among its specifications:

- A subquery (but the grid query itself can be a subquery of an outer query that it does not reference)
- A join operation across grid servers
- Connection requests that result in cross-server joins
- A procedure or function that does not exist on all participating grid servers.

Neither the UNION nor UNION ALL set operators, nor the INTERSECT, MINUS, or EXCEPT set operators, are valid in a grid query block.

The GRID clause should not be included in SELECT statements outside a grid context. For more information about grids, see the #unique_981 and the .

WHERE clause of SELECT

The WHERE clause can specify join conditions for Informix®-extension joins, post-join filters for ANSI-compliant joins, and for search criteria on data values.

This syntax fragment is part of the SELECT statement on page 690.

(explicit id unique_988_Connect_42_whcl004) unique_988_Connect_42_whcl004(explicit id unique_988_Connect_42_whcl005)
unique_988_Connect_42_whcl005(explicit id unique_988_Connect_42_whcl006) unique_988_Connect_42_whcl006(explicit id unique_988_Connect_42_whcl007)
unique_988_Connect_42_whcl007(explicit id unique_988_Connect_42_whcl008) unique_988_Connect_42_whcl008

WHERE Clause

" **WHERE** "

" { | <Condition> [] | <Join> [] | <Function Expression> [] | (*subquery*) | { (<Collection Subquery> []) |
<Statement Local Variable Expressions> [] } } "

Element	Description	Restrictions	Syntax
Logical_ Operator	Combines two conditions	Valid options are *logical union* (= OR *or* OR NOT) or *logical intersection* (= AND *or* AND NOT)	Conditions with AND or OR on page 984
subquery	Embedded query	Cannot include the FIRST or ORDER BY keywords	SELECT statement on page 690

Using a Condition in the WHERE Clause

You can use these simple conditions or comparisons in the WHERE clause:

- Relational-operator condition
- IN or BETWEEN . . . AND
- IS NULL or IS NOT NULL
- LIKE or MATCHES

You also can use a SELECT statement within the WHERE clause; this is called a *subquery*. The following WHERE clause operators are valid in a subquery:

- IN or EXISTS
- ALL, ANY, or SOME

For more information, see Condition on page 967.

In the WHERE clause, an aggregate function is not valid unless it is part of a subquery or is on a correlated column originating from a parent query, and the WHERE clause is in a subquery within a HAVING clause.

Relational-Operator Condition

A relational-operator condition is satisfied if the expressions on each side of the operator fulfill the relation that the operator specifies. The following statements use the greater than (>) and equal (=) relational operators:

```
SELECT order_num FROM orders
   WHERE order_date > '6/04/08';
SELECT fname, lname, company
   FROM customer
   WHERE city[1,3] = 'San';
```

Single quotation marks are required around '`San`' because the substring is from a character column. See the Relational-Operator Condition on page 970.

Blank strings and empty strings in the WHERE clause

For LVARCHAR, NVARCHAR, or VARCHAR columns, queries with a WHERE clause specifying equality of the column value to an empty string (

```
WHERE varlength_col = ''
```

) return the same result set as an otherwise identical query in which the WHERE clause specifies equality to a string of blank (ASCII 32) characters.

For example, if `varlength_col` is of type VARCHAR, NVARCHAR, or LVARCHAR, the following WHERE clause examples are all functionally equivalent to a WHERE clause that specifies equality to an empty string:

```
   WHERE varlength_col = ' '
   WHERE varlength_col = '  '
   WHERE varlength_col = '   '
```

Thus, for the built-in variable-length character data types, the WHERE clause makes no distinction between an empty string and a string consisting entirely of one or more blank characters. (Note, however, that the query filter

```
   WHERE varlength_col IS NULL
```

is not equivalent to the previous WHERE clause examples, and returns a different result set if any `varlength_col` value is NULL.)

IN Condition

The IN condition is satisfied when the expression to the left of the IN keyword is included in the list of values to the right of the keyword.

The following examples show the IN condition:

```
SELECT lname, fname, company FROM customer
   WHERE state IN ('CA','WA', 'NJ');
SELECT * FROM cust_calls
   WHERE user_id NOT IN (USER );
```

For more information, see the IN Subquery on page 981.

BETWEEN Condition

The BETWEEN condition is satisfied when the value to the left of BETWEEN is in the inclusive range of the two values on the right of BETWEEN. The first two queries in the following example use literal values after the BETWEEN keyword. The third

query uses the built-in CURRENT function and a literal interval to search for dates between the current day and seven days earlier.

```
SELECT stock_num, manu_code FROM stock
   WHERE unit_price BETWEEN 125.00 AND 200.00;
SELECT DISTINCT customer_num, stock_num, manu_code
   FROM orders, items
   WHERE order_date BETWEEN '6/1/07' AND '9/1/07';
SELECT * FROM cust_calls WHERE call_dtime
   BETWEEN (CURRENT - INTERVAL(7) DAY TO DAY) AND CURRENT;
```

For more information, see the BETWEEN Condition on page 971.

Using IS NULL and IS NOT NULL Conditions

The IS NULL condition is satisfied if the specified *column* contains a NULL value, or if the specified *expression* evaluates to NULL.

If you use the IS NOT NULL predicate, the condition is satisfied when the *column* contains a value that is not NULL, or when the *expression* does not evaluate to NULL. The following example selects the order numbers and customer numbers for which the order has not been paid:

```
SELECT order_num, customer_num FROM orders
   WHERE paid_date IS NULL;
```

For a complete description of the IS NULL and IS NOT NULL operators, see the IS NULL and IS NOT NULL Conditions on page 974.

LIKE or MATCHES Condition

The LIKE or MATCHES condition is satisfied if either of the following is true:

- The value of the column that precedes the LIKE or MATCHES keyword matches the pattern that the quoted string specifies. You can use wildcard characters in the string.
- The value of the column that precedes the LIKE or MATCHES keyword matches the pattern that is specified by the column that follows the LIKE or MATCHES keyword. The value of the column on the right serves as the matching pattern in the condition.

The following examples use a backslash (****) as the default escape character. The default escape character is set by the DEFAULTESCCHAR configuration parameter or the **DEFAULTESCCHAR** session environment option.

The following SELECT statement returns all rows in the **customer** table in which the **lname** column begins with the literal string `'Baxter'`. Because the string is a literal string, the condition is case sensitive.

```
SELECT * FROM customer WHERE lname LIKE 'Baxter%' ;
```

The next SELECT statement returns all rows in the **customer** table in which the value of the **lname** column matches the value of the **fname** column:

```
SELECT * FROM customer WHERE lname LIKE fname;
```

The following examples use the LIKE condition with a wildcard. The first SELECT statement finds all stock items that are some kind of ball. The second SELECT statement finds all company names that contain a percent (%) sign. Backslash (****) is used as the default escape character for the percent (%) sign wildcard. The third SELECT statement uses the ESCAPE option with the LIKE condition to retrieve rows from the **customer** table in which the **company** column includes a percent (%) sign. The `z` is used as an escape character for the percent (%) sign:

```
SELECT stock_num, manu_code FROM stock
   WHERE description LIKE '%ball';
SELECT * FROM customer WHERE company LIKE '%\%%';
SELECT * FROM customer WHERE company LIKE '%z%%' ESCAPE 'z';
```

The following examples use MATCHES with a wildcard in SELECT statements. The first SELECT statement finds all stock items that are some kind of ball. The second SELECT statement finds all company names that contain an asterisk (*****). The backslash (****) is used as the default escape character for a literal asterisk (*****) character. The third statement uses the ESCAPE option with the MATCHES condition to retrieve rows from the **customer** table where the **company** column includes an asterisk (*****). The `z` character is specified as an escape character for the asterisk (*****) character:

```
SELECT stock_num, manu_code FROM stock
   WHERE description MATCHES '*ball';

SELECT * FROM customer WHERE company MATCHES '*\**';

SELECT * FROM customer WHERE company MATCHES '*z**' ESCAPE 'z';
```

For information about the supported data types of operands in LIKE or MATCHES expressions, see the topic LIKE and MATCHES Condition on page 975.

IN Subquery

With the IN subquery, more than one row that satisfies the IN or NOT IN condition can be returned, but only one column can be returned.

This example shows the use of a NOT IN subquery in a SELECT statement:

```
SELECT DISTINCT customer_num FROM orders
   WHERE order_num NOT IN
      (SELECT order_num FROM items
         WHERE stock_num = 1);
```

For additional information, see the IN Condition on page 972.

EXISTS Subquery

From the EXISTS subquery, rows that satisfy EXISTS conditions in one or more columns can be returned. (Similarly, the NOT EXISTS subquery can returns rows that satisfy NOT EXISTS conditions in one or more columns.)

The following example of a SELECT statement with a NOT EXISTS subquery returns the stock number and manufacturer code for every item that has never been ordered (and is therefore not listed in the **items** table).

It is appropriate to use a NOT EXISTS subquery in this SELECT statement because you need the correlated subquery to test both **stock_num** and **manu_code** in the **items** table.

```
SELECT stock_num, manu_code FROM stock
   WHERE NOT EXISTS
      (SELECT stock_num, manu_code FROM items
         WHERE stock.stock_num = items.stock_num AND
            stock.manu_code = items.manu_code);
```

The preceding example would work equally well if you use a SELECT * in the subquery in place of the column names, because you are testing for the existence of a row or rows.

For additional information, see the EXISTS Subquery condition on page 982.

ALL, ANY, SOME Subqueries

The following examples return the order number of all orders that contain an item whose total price is greater than the total price of every item in order number 1023. The first SELECT uses the ALL subquery, and the second SELECT produces the same result by using the MAX aggregate function.

```
SELECT DISTINCT order_num FROM items
   WHERE total_price > ALL (SELECT total_price FROM items
        WHERE order_num = 1023);

SELECT DISTINCT order_num FROM items
   WHERE total_price > SELECT MAX(total_price) FROM items
        WHERE order_num = 1023);
```

The following SELECT statements return the order number of all orders that contain an item whose total price is greater than the total price of at least one of the items in order number 1023. The first SELECT statement uses the ANY keyword, and the second SELECT statement uses the MIN aggregate function:

```
SELECT DISTINCT order_num FROM items
   WHERE total_price > ANY (SELECT total_price FROM items
        WHERE order_num = 1023);

SELECT DISTINCT order_num FROM items
   WHERE total_price > (SELECT MIN(total_price) FROM items
      WHERE order_num = 1023);
```

You can omit the keywords ANY, ALL, or SOME in a subquery if the subquery returns exactly one value. If you omit ANY, ALL, or SOME, and the subquery returns more than one value, you receive an error. The subquery in the next example returns only one row, because it uses an aggregate function:

```
SELECT order_num FROM items
   WHERE stock_num = 9 AND quantity =
      (SELECT MAX(quantity) FROM items WHERE stock_num = 9);
```

See also ALL, ANY, and SOME Subqueries on page 982.

Specifying a Join in the WHERE Clause

You join two tables by creating a relationship in the WHERE clause between at least one column from one table and at least one column from another. The join creates a temporary composite table where each pair of rows (one from each table) that satisfies the join condition is linked to form a single row.

This syntax fragment is part of the WHERE clause of SELECT on page 731.

(explicit id unique_1032_Connect_42_specj002) unique_1032_Connect_42_specj002

Join

" [<Data Source>] "

" *column* "

" <Relational Operator> [] "

" [<Data Source>] "

" *column* "

Data Source

" { *alias* . | *external* . | *table* . | *view* . | *synonym* . } "

Element	Description	Restrictions	Syntax
alias	Temporary alternative name declared in the FROM clause for a table or view	See the Self joins section; FROM Clause on page 711	Identifier on page 1231
column	Column of a table or view to be joined	Must exist in the table or view	Identifier on page 1231
external	External table from which to retrieve data	External table must exist. Must include appropriate qualifiers.	Database Object Name on page 1225
synonym, *table*, *view*	Name of a synonym, table, or view to be joined in the query	Synonym and the table or view to which it points must exist. Must include appropriate qualifiers.	Database Object Name on page 1225

Rows from the tables or views are *joined* when there is a match between the values of specified columns. When the columns to be joined have the same name, you must qualify each column name with its data source. You can create two-table joins, multiple-table joins, self-joins, and outer joins (Informix®-extension syntax).

Two-table joins

The following example shows a two-table join:

```
SELECT order_num, lname, fname FROM customer, orders
   WHERE customer.customer_num = orders.customer_num;
```

Multi-table joins

A multiple-table join is a join of more than two tables. Its structure is similar to the structure of a two-table join, except that you have a join condition for more than one pair of tables in the WHERE clause. When columns from different tables have the same name, you must qualify the column name with its associated table or table alias, as in *table.column*.

The following multiple-table join yields the company name of the customer who ordered an item as well as its stock number and manufacturer code:

```
SELECT DISTINCT company, stock_num, manu_code
   FROM customer c, orders o, items i
   WHERE c.customer_num = o.customer_num
      AND o.order_num = i.order_num;
```

Self joins

You can join a table to itself. To do so, you must list the table name twice in the FROM clause and assign it two different table aliases. Use the aliases to refer to each of the *two* tables in the WHERE clause. The next example is a self-join on the **stock** table. It finds pairs of stock items whose unit prices differ by a factor greater than 2.5. The letters `x` and `y` are each aliases for the **stock** table.

```
SELECT x.stock_num, x.manu_code, y.stock_num, y.manu_code
   FROM stock x, stock y WHERE x.unit_price > 2.5 * y.unit_price;
```

Informix®-extension outer joins

The next outer join lists the company name of the customer and all associated order numbers, if the customer has placed an order. If not, the company name is still listed, and a NULL value is returned for the order number.

```
SELECT company, order_num FROM customer c, OUTER orders o
   WHERE c.customer_num = o.customer_num;
```

For the syntax of outer joins, see Informix-Extension Outer Joins on page 726.

Hierarchical Clause

The Hierarchical clause sets the conditions for recursive queries on a table object in which a hierarchy of parent-child dependencies exists among the rows. SELECT statements that include this clause are called *hierarchical queries*.

The table object on which the hierarchical query operates must be specified in the FROM clause of the SELECT statement. The table object is typically a self-referencing table in which one or more columns acts as a foreign key constraint for another column (or for a subset of the columns) in the same table.

A hierarchical query operates on rows in which one or more column values correspond to nodes within a logical structure of parent-child relationships. If parent rows have multiple children, sibling relationships exist among child rows of the same parent. These relationships might reflect, for example, the reporting structure among employees and managers within the divisions and management levels of an organization.

The syntax that this clause supports is an extension to the ANSI/ISO standard for SQL. This syntax fragment is part of the SELECT statement on page 690.

Hierarchical Clause

(explicit id unique_482_Connect_42_hierarch001) unique_482_Connect_42_hierarch001 (explicit id unique_482_Connect_42_hierarch002) unique_482_Connect_42_hierarch002

" [<START WITH Clause>] "

" < CONNECT BY Clause> "

The table object on which the hierarchical query operates must be specified in the FROM clause of the SELECT statement. The table object can be any of the following table objects:

- A table or updateable view
- A temporary table
- A table in another database of the same Informix® instance to which the session is connected
- A derived table that is the result of a query
- A table that is protected by a label-based access control (LBAC) security policy
- A table with column level encryption or row level encryption
- A synonym for any of the other table objects.

The following table objects are not supported in the FROM clause of a hierarchical query:

- A join of two or more tables
- A view that is not updatable
- A table in a database of a remote Informix® instance
- An external table that the CREATE EXTERNAL TABLE statement defined
- A sequence object.

Informix® supports sequence objects in the projection list of hierarchical queries, in the WHERE clause, and in other contexts where an expression is valid in SELECT statements, but not in the hierarchical query clause.

The hierarchical clause is valid in correlated subqueries and in uncorrelated subqueries.

Hierarchical queries can include all types of optimizer directives, with these exceptions:

- Join-order directives
- Join-method directives

Hierarchical queries do not support the Parallel Database Query (PDQ) feature of Informix®.

The Hierarchical clause can specify recursive queries on a table whose rows describe a hierarchy of parent-child relationships.

- The hierarchy can be a simple hierarchy, such as the reporting structure of an organization in which every node that is not the root reports to a single node at higher level within the hierarchy. (In the LBAC security feature of Informix®, a security label component of type TREE has the logical structure of a simple hierarchy.)
- The Hierarchical clause can query data hierarchies of more complex topologies, in which nodes have many-to-many relationships, and in which a child node can be the ancestor of its parent. For information about using the Hierarchical clause to query a table that has cycles within the data hierarchy, see CONNECT BY Clause on page 744.

Important: Hierarchical queries are most efficient for data sets in which parent-child dependencies in the table have the logical topology of a simple graph. If the self-referencing table includes more than one independent hierarchy for the same set of columns, or if any child row is also an ancestor of its parent, see Dependency patterns that are not a simple graph on page 752.

Note: The Hierarchical clause is unrelated to table hierarchies, in which a hierarchy of parent-child relationships exist among the schemas of a set of typed tables. Similarly, the hierarchy of a set of DISTINCT data types that all derive from a common base type resembles a data hierarchy, but is unrelated to the Hierarchical clause, where the hierarchy exists in parent-child dependencies between data entities, rather than relationships among data types.

SQL Syntax Specific to Hierarchical Queries

Besides the START WITH, CONNECT BY, and CONNECT NOCYCLE BY keywords that specify the conditions for recursive queries of a table that contains hierarchical data, hierarchical queries also support syntax tokens that are valid only in hierarchical queries, and that cannot be used in SELECT statements that have no CONNECT BY clause. Syntax tokens specific to hierarchical queries include two operators, three pseudocolumns, and a built-in function:

- CONNECT_BY_ROOT operator

  This operator can return an expression for the root ancestor of its operand.

- PRIOR operator

  This operator can reference a returned value from the previous recursive step (where "step" refers to an iteration of the recursive query).

- LEVEL pseudocolumn

  This pseudocolumn returns an integer, indicating which step of the recursive query returned a row within the hierarchy.

- CONNECT_BY_ISCYCLE pseudocolumn

  This pseudocolumn can indicate whether a row has a child row that is also its ancestor.

- CONNECT_BY_ISLEAF pseudocolumn

  This pseudocolumn can indicate whether a row has any children among the rows that the query returns.

- SYS_CONNECT_BY_PATH function

  This function can construct and return a string that represents the path from a specified row to the root of the hierarchy

- SIBLINGS keyword in the ORDER BY clause

  The ORDER SIBLINGS BY clause can sort returned rows for siblings of the same parent at every level.

A *pseudocolumn* is a built-in identifier that the SQL parser can recognize in specific contexts, and that shares the same namespace as columns and variables. These pseudocolumns and the SYS_CONNECT_BY_PATH function are typically specified in the Projection clause of the SELECT statement, but the LEVEL pseudocolumn and the PRIOR operator can be specified in the Hierarchical clause.

For details of the syntax and semantics of these tokens that support only hierarchical queries, see Conditions in the CONNECT BY Clause on page 746 and ORDER SIBLINGS BY Clause on page 766.

Overview of Hierarchical Queries

The clauses of a SELECT statement that includes the Hierarchical clause are processed in the following sequence:

1. FROM clause (for only a single table object in the current database)
2. Hierarchical clause
3. WHERE clause (without join predicates)
4. GROUP BY clause
5. HAVING clause
6. Projection clause
7. ORDER BY clause

The ORDER BY SIBLING option of the ORDER BY clause can order the set of rows that are children of the same parent.

A subquery that includes the Hierarchical clause returns the intermediate result set in a partial order, where the rows produced in iteration (n+1) for a specific hierarchy immediately follow the row in iteration (n) that produced them. However, specifying an ORDER BY clause, a GROUP BY or HAVING clause, or the DISTINCT or UNIQUE keyword in the Projection clause destroys that partial order.

The Hierarchical clause follows the WHERE clause in the lexical sequence of SELECT statement clauses, but the WHERE clause predicates are processed on the result of the Hierarchical clause. The WHERE clause cannot specify join predicates if the SELECT statement includes the Hierarchical clause, but the table object that is specified in the FROM clause can be the result set of a query that joins one or more tables.

Any SELECT statement that includes a hierarchical-query-clause is called a hierarchical query, which performs a recursive series of queries on the table that the FROM clause specifies:

1. The optional START WITH clause can specify a condition. Any rows that satisfy this condition are returned as the first intermediate result set of the hierarchical query.
2. The next step applies the condition that is specified in the CONNECT BY clause to the table. Any rows that satisfy that condition are returned as the second intermediate result set.
3. The next step applies the CONNECT BY condition to the table. Any rows that are returned comprise the third intermediate result set.
4. The CONNECT BY clause runs queries recursively to produce successive intermediate result sets, until an iteration yields an empty result set.
5. The hierarchical SELECT statement then combines all of the intermediate result sets of the preceding recursive steps, producing the final result set of the Hierarchical clause.
6. The predicates of the WHERE clause are then applied to this set of rows that the Hierarchical clause retrieved, and the remaining clauses of the SELECT statement are then applied in the order listed.

After the START WITH and CONNECT BY clauses return all of the intermediate result sets, you can use the ORDER SIBLINGS BY clause to sort the sibling rows that have the same parent for every level within the hierarchy. For more information, see ORDER SIBLINGS BY Clause on page 766.

You can use output from the SET EXPLAIN statement to view the execution path of a hierarchical query.

The Hierarchical clause provides an efficient alternative to using the Node database extension to retrieve information from hierarchical data sets

Example of a Hierarchical Data Set

In several topics that follow, SQL code examples that illustrate hierarchical queries are based on hierarchic data in the following **employee** table, whose rows contains information about employees within an organizational hierarchy. The **mgrid** column shows the employee identifier (**empid**) of the manager to whom the employee reports:

```
CREATE TABLE employee(
                      empid  INTEGER NOT NULL PRIMARY KEY,
                      name   VARCHAR(10),
                      salary DECIMAL(9, 2),
                      mgrid  INTEGER
);
```

Data values for the 17 rows in the **employee** table are these.

```
INSERT INTO employee VALUES  ( 1, 'Jones',    30000, 10);
INSERT INTO employee VALUES  ( 2, 'Hall',     35000, 10);
INSERT INTO employee VALUES  ( 3, 'Kim',      40000, 10);
INSERT INTO employee VALUES  ( 4, 'Lindsay',  38000, 10);
INSERT INTO employee VALUES  ( 5, 'McKeough', 42000, 11);
INSERT INTO employee VALUES  ( 6, 'Barnes',   41000, 11);
INSERT INTO employee VALUES  ( 7, 'O''Neil',  36000, 12);
INSERT INTO employee VALUES  ( 8, 'Smith',    34000, 12);
INSERT INTO employee VALUES  ( 9, 'Shoeman',  33000, 12);
INSERT INTO employee VALUES  (10, 'Monroe',   50000, 15);
INSERT INTO employee VALUES  (11, 'Zander',   52000, 16);
INSERT INTO employee VALUES  (12, 'Henry',    51000, 16);
INSERT INTO employee VALUES  (13, 'Aaron',    54000, 15);
INSERT INTO employee VALUES  (14, 'Scott',    53000, 16);
```

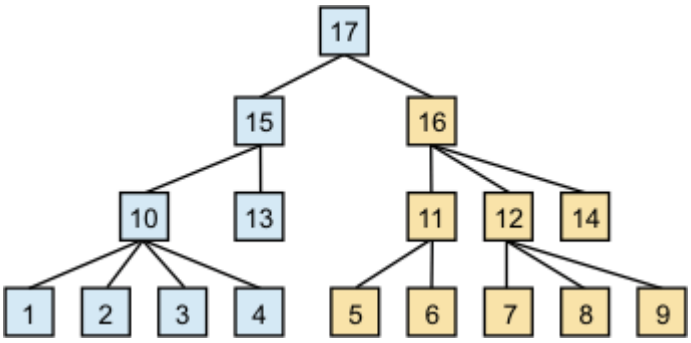
```
INSERT INTO employee VALUES  (15, 'Mills',    70000, 17);
INSERT INTO employee VALUES  (16, 'Goyal',    80000, 17);
INSERT INTO employee VALUES  (17, 'Urbassek', 95000, NULL);
```

Each pair of **empid** and **mgrid** values express referential relationships that the recursive iterations of a query with an appropriate CONNECT BY condition can correctly assemble into a hierarchy.

Here the NULL value in the **mgrid** column in the last row shows that employee **Urbassek**, whose **empid** value is `17` is the root node of this reporting hierarchy.

The following diagram illustrates the four levels of the reporting hierarchy (with nodes that show the **empid** values) for the **employee** table data:

Figure 9. Relationships of Elements in a Reporting Hierarchy



START WITH Clause

The optional START WITH clause specifies a condition. The row that satisfies this condition becomes the root for beginning the recursive operations of the CONNECT BY clause in hierarchical queries.

The START WITH clause is an extension to the ANSI/ISO standard for SQL. This syntax fragment is part of the Hierarchical Clause on page 738.

START WITH Clause

(explicit id unique_1062_Connect_42_startif002) unique_1062_Connect_42_startif002

```
" START WITH <Condition> [] "
```

Usage

The START WITH clause specifies a search condition that the CONNECT BY clause uses for the first iteration of its recursive actions. If you omit the START WITH clause, the CONNECT BY clause treats every row as the root of the hierarchy for the initial set of intermediate results.

CONNECT BY Clause

The CONNECT BY clause specifies conditions for performing recursive operations in hierarchical queries.

The CONNECT BY clause is an extension to the ANSI/ISO standard for SQL. This syntax fragment is part of the Hierarchical Clause on page 738.

CONNECT BY Clause

(explicit id unique_1063_Connect_42_cby002) unique_1063_Connect_42_cby002

```
" CONNECT BY [NOCYCLE]  <Condition> [] "
```

Usage

If you include the START WITH clause, the search condition that it specifies is applied in producing the first intermediate result set for the hierarchical query. This consists of the rows of the table specified in the FROM clause for which the START WITH condition is true.

If the START WITH clause is omitted, no START WITH condition is available as a filter, and the first intermediate result set is the entire set of rows in the table that the FROM clause specifies.

The CONNECT BY clause produces successive intermediate result sets by applying the CONNECT BY search condition until this recursive process terminates when an iteration yields an empty result set.

The NOCYCLE Keyword

Rows returned by recursive queries of the CONNECT BY clause must be part of a simple hierarchy. SELECT statements that include the Hierarchical clause fail with an error if the query returns a row that is both the ancestor and the descendant of another node. This topology is called a cycle.

You can include the NOCYCLE keyword between the CONNECT BY keywords and the condition specification of the CONNECT BY clause to filter out any rows that would otherwise cause the hierarchical query to fail with error -26079 because of a cycle in an intermediate result set.

For example, for the hierarchical data set of the **employee** table that is described in the topicHierarchical Clause on page 738, the following UPDATE statement introduces a cycle for the employees whose **empid** values are 5 and 17:

```
UPDATE employee SET mgrid = 5 WHERE name = 'Urbassek';
```

After the hierarchical data set has been modified by the UPDATE statement above, the following query (which omits the NOCYCLE keyword) fails:

```
SELECT empid, name, mgrid , CONNECT_BY_ISLEAF leaf
FROM employee
START WITH name = 'Goyal'
CONNECT BY PRIOR empid = mgrid;
```

Error -26079 is issued when the last CONNECT BY step detects that employee **McKeough** is part of a loop:

```
      empid name             mgrid   leaf

         16 Goyal               17      0
         14 Scott               16      1
         12 Henry               16      0
          9 Shoeman             12      1
          8 Smith               12      1
          7 O'Neil              12      1
         11 Zander              16      0
          6 Barnes              11      1
          5 McKeough            11      0

26079: CONNECT BY query resulted in a loop/cycle.
Error in line 8
Near character position 28
```

You can include the NOCYCLE keyword between the CONNECT BY keywords and the *condition* specification of the CONNECT BY clause to filter out any rows that would otherwise cause the hierarchical query to fail with error -26079 because of a cycle in an intermediate result set. The following query differs from the query that failed by including the CONNECT_BY_ISCYCLE pseudocolumn in the Projection clause, and by including the NOCYCLE keyword in the CONNECT BY clause.

```
SELECT empid, name, mgrid, CONNECT_BY_ISLEAF leaf, CONNECT_BY_ISCYCLE cycle
FROM employee
START WITH name = 'Goyal'
CONNECT BY NOCYCLE PRIOR empid = mgrid;

      empid name             mgrid   leaf  cycle

         16 Goyal               17      0      0
         14 Scott               16      1      0
         12 Henry               16      0      0
          9 Shoeman             12      1      0
          8 Smith               12      1      0
          7 O'Neil              12      1      0
         11 Zander              16      0      0
          6 Barnes              11      1      0
          5 McKeough            11      0      0
         17 Urbassek             5      0      1
         15 Mills               17      0      0
         13 Aaron               15      1      0
         10 Monroe              15      0      0
          4 Lindsay             10      1      0
          3 Kim                 10      1      0
          2 Hall                10      1      0
          1 Jones               10      1      0

17 row(s) retrieved.
```

Because the NOCYCLE keyword enabled the CONNECT BY clause to continue processing after the cycle was detected, **Urbassek** was returned from the CONNECT BY step that had failed in the previous example, and processing continued until all of the rows in the data set had been returned. In the output display above, **leaf** is an alias for the CONNECT_BY_ISLEAF pseudocolumn, and **cycle** is an alias for the CONNECT_BY_ISCYCLE pseudocolumn, with both aliases declared in the Projection clause. In these results, **Urbassek** is marked in the **cycle** column as the cause of the loop.

The result set above implies that the cycle can be removed from the **employee** table by changing the **mgrid** value in the row that had identified **McKeough** as the manager of **Urbassek**:

```
UPDATE employee SET mgrid = NULL WHERE empid = 17;
```

Conditions in the CONNECT BY Clause

Besides expressions and operators that are valid in Boolean conditions and in general SQL expressions, the *condition* that is specified in CONNECT BY clause supports two more syntax constructs, the PRIOR operator and the LEVEL pseudocolumn that are valid only in SELECT statements that include the Hierarchical clause.

The PRIOR Operator

The PRIOR unary operator can be included in the CONNECT BY clause with a column name as its operand. PRIOR can be used to distinguish column references to the result of the most recent previous recursive step of the CONNECT BY clause from column references to the current result set. The column name immediately follows this right-associative operator, as in the following syntax fragment:

```
 CONNECT BY mgrid = PRIOR empid
```

Here the CONNECT BY condition is satisfied by those rows in which the manager specified in the **mgrid**, column matches the employee value was in the **empid** column in the previous iteration.

The PRIOR operator can be applied to expressions more complex than column names. The following condition uses an arithmetic expression as the operand of PRIOR:

```
 CONNECT BY PRIOR (salary - 10000) = salary
```

The PRIOR operator can be included more than once in the same CONNECT BY condition. See also the topic Hierarchical Clause on page 738, which provides an example of a hierarchical query that uses the PRIOR operator in a condition of the CONNECT BY clause.

The LEVEL Pseudocolumn

A *pseudocolumn* is a keyword of SQL that shares the same namespace as column names, and that is valid in some contexts where a column expression is valid.

LEVEL is a pseudocolumn that returns the ordinal number of the recursive step in the Hierarchic clause that returned the row. For all the rows returned by the START WITH clause, LEVEL return the value `1`. Rows that are returned by applying the first iteration of the CONNECT BY clause return 2. Rows that are returned by successive iterations of the CONNECT BY have LEVEL values incremented by 1, so that LEVEL = (N + 1) indicates a row that the Nth CONNECT BY iteration returned. The data type of the LEVEL column is INTEGER.

The following example of a hierarchical query specifies LEVEL in the select list of the Projection clause:

```
SELECT name, LEVEL FROM employee START WITH name = 'Goyal'
   CONNECT BY PRIOR empid = mgrid;
```

The query returns these results:

```

name          level
```

```
Goyal             1
Zander            2
McKeough          3
Barnes            3
Henry             2
O'Neil            3
Smith             3
Shoeman           3
Scott             2

9 row(s) retrieved.
```

LEVEL can be included in the Projection clause of SELECT statements that include the Hierarchical clause, and in the *condition* of the CONNECT BY clause.

The LEVEL pseudocolumn is not valid, however, in the following contexts:

- A SELECT statement that has no CONNECT BY clause
- The START WITH *condition* of the Hierarchical clause
- An operand of the CONNECT_BY_ROOT operator
- An argument to the SYS_CONNECT_BY_PATH function.

Additional Syntax Valid Only in Hierarchical Queries

The following syntax tokens support hierarchical queries, and are valid only in hierarchical queries. Unlike the PRIOR operator and the LEVEL pseudocolumn, however, they are not valid in the Hierarchical clause:

- The CONNECT_BY_ISCYCLE pseudocolumn
- The CONNECT_BY_ISLEAF pseudocolumn
- The CONNECT_BY_ROOT unary operator
- The SYS_CONNECT_BY_PATH( ) function of SQL.

The CONNECT_BY_ISCYCLE Pseudocolumn

CONNECT_BY_ISCYCLE is a pseudocolumn that returns a `1` if the row would cycle at the next level in the hierarchy. That is, the row has an immediate child that is also an ancestor given the search-condition that is specified in the CONNECT BY clause. If the row does not directly cause a cycle, the column returns `0`. A value other than `0` is only possible when NOCYCLE is specified in the CONNECT BY clause. The data type of this column is INTEGER.

The following UPDATE statement creates a loop in the data hierarchy of the **employee** table:

```
UPDATE employee SET mgrid = 5 WHERE empid = 17;
```

The following hierarchical query includes the CONNECT_BY_ISCYCLE pseudocolumn in the Projection clause, but the CONNECT BY clause throws an error in the step where it encounters the loop that the UPDATE statement created.

```
SELECT empid,
       name,
       mgrid,
       CONNECT_BY_ISLEAF leaf,
```

```
       CONNECT_BY_ISCYCLE cycle
FROM  employee
START WITH name = 'Goyal'
CONNECT BY  PRIOR empid = mgrid;

  665: Internal error on semantics -
  CONNECT_BY_ISCYCLE is used without NOCYCLE parameter..
Error in line 1
Near character position 72
```

This query avoids the -655 error by specifying the NOCYCLE in the CONNECT BY clause that throws an error in the step where it encounters the loop that the UPDATE statement created.

```
SELECT empid, name, mgrid,
   CONNECT_BY_ISLEAF leaf, CONNECT_BY_ISCYCLE cycle
FROM employee
   START WITH name = 'Goyal'
   CONNECT BY NOCYCLE PRIOR empid = mgrid;
```

For the results of this query, see the example in the description of the NOCYCLE keyword in the topic CONNECT BY Clause on page 744.

The CONNECT_BY_ISCYCLE pseudocolumn is not valid in the following contexts:

- A SELECT statement that has no CONNECT BY clause
- The START WITH or CONNECT BY clause
- An operand of the CONNECT_BY_ROOT operator
- An argument to the SYS_CONNECT_BY_PATH function

The CONNECT_BY_ISLEAF Pseudocolumn

CONNECT_BY_ISLEAF is a pseudocolumn that returns a `1` if the row is a leaf in the hierarchy as defined by the CONNECT BY clause. A node is a *leaf node* if it has no children in the query result hierarchy (not in the actual data hierarchy). If the row is not a leaf the column returns `0`. The data type of the column is INTEGER.

The following hierarchical query specifies the CONNECT_BY_ISLEAF pseudocolumn in the Projection clause, and declares **leaf** as an alias for that column, which shows in the DB-Access display of the result set:

```
SELECT empid, name, mgrid, CONNECT_BY_ISLEAF leaf
FROM emp1oyee
START WITH name = 'Goyal'
CONNECT BY PRIOR empid = mgrid;

      empid name             mgrid   leaf

         16 Goyal               17      0
         14 Scott               16      1
         12 Henry               16      0
          9 Shoeman             12      1
          8 Smith               12      1
          7 O'Neil              12      1
         11 Zander              16      0
          6 Barnes              11      1
```

```
          5 McKeough            11      1

9 row(s) retrieved.
```

The CONNECT_BY_ROOT Operator

For every row in the hierarchy, the CONNECT_BY_ROOT unary operator accepts as its operand an expression that evaluates to a row that is a node of the hierarchy. CONNECT_BY_ROOT returns the expression for the root ancestor of its operand. This syntax fragment is part of the Hierarchical Clause on page 738.

CONNECT_BY_ROOT Operator

> " **CONNECT_BY_ROOT** *expression* "

The *expression* operand can be any SQL expression, but it must not contain any hierarchical query token, including the following tokens:

- The CONNECT_BY_ROOT or PRIOR unary operators
- The CONNECT_BY_ISCYCLE, CONNECT_BY_ISLEAF, or LEVEL pseudocolumns
- The SYS_CONNECT_BY_PATH function.

The return data type of this right-associative operator is the data type of the specified expression.

The hierarchical query in the following example returns rows from the **employee** table that include both the identifying number of the manager to whom the employee reports directly, and also the name of the manager at the root of the hierarchy for this query.

```
SELECT empid, name, mgrid, CONNECT_BY_ROOT name AS topboss
FROM employee
START WITH name = 'Goyal'
CONNECT BY PRIOR empid = mgrid;

      empid name             mgrid topboss

         16 Goyal               17 Goyal
         14 Scott               16 Goyal
         12 Henry               16 Goyal
          9 Shoeman             12 Goyal
          8 Smith               12 Goyal
          7 O'Neil              12 Goyal
         11 Zander              16 Goyal
          6 Barnes              11 Goyal
          5 McKeough            11 Goyal

9 row(s) retrieved.
```

The CONNECT_BY_ROOT operator is not valid in the following contexts:

- A SELECT statement that has no CONNECT BY clause
- The START WITH or CONNECT BY clause
- An argument to the SYS_CONNECT_BY_PATH function

The SYS_CONNECT_BY_PATH Function

Calls to SYS_CONNECT_BY_PATH ( ) function are valid in SELECT statements that include the Hierarchical clause, but this function cannot be called from the Hierarchical clause. Informix® returns an error if you attempt to run this function within the *condition* of the START WITH or CONNECT BY clauses.

The SYS_CONNECT_BY_PATH function can be used in hierarchical queries to build a string that represents a path from the row corresponding to the root node to the current row.

This is the calling syntax for SYS_CONNECT_BY_PATH to return a string for a specified row at LEVEL `N`. This syntax fragment is part of the Hierarchical Clause on page 738.

SYS_CONNECT_BY_PATH Function

" `SYS_CONNECT_BY_PATH` (*string_expression* , ' *format_string* ') "

Element	Description	Restrictions	Syntax
format_string	Typically a constant string that serves as a separator	None	Quoted String on page 1202
string_expression	An expression that identifies a row.	Cannot include hierarchical query tokens	Expression on page 1003

SYS_CONNECT_BY_PATH builds the string representation of the path from the root to a specified row at LEVEL N of the hierarchy by recursively concatenating the successive returned values:

- *path1* := *string_expression1*||*format_string* represents the path to the root row from the first intermediate result set,
- *path2* := *path1*||*string_expression2*||*format_string* evaluates to the path from the root to a row in the second intermediate result set,
- . . .
- *pathN* := *path(N-1)*||*string_expressionN*||*format_string* evaluates to the path from the root to the Nth intermediate result set.

The expressions in arguments to SYS_CONNECT_BY_PATH must not include any hierarchical query construct, including the following constructs:

- The CONNECT_BY_ROOT or PRIOR unary operators
- The CONNECT_BY_ISCYCLE, CONNECT_BY_ISLEAF, or LEVEL pseudocolumns
- The SYS_CONNECT_BY_PATH function.

Also not valid in the argument list are aggregate expressions.

The return value from SYS_CONNECT_BY_PATH ( ) is of type LVARCHAR(4000).

The hierarchical query in the following example calls the SYS_CONNECT_BY_PATH function in the Projection list with the **employee.name** column and the slash (/) character as its arguments.

```
SELECT empid, name, mgrid, SYS_CONNECT_BY_PATH( name,'/') as hierarchy
   FROM employee
   START WITH name = 'Henry'
   CONNECT BY PRIOR empid = mgrid;
```

The query returns the rows within the subset of the data hierarchy in which **Henry** is specified as the root in the START WITH clause, showing the name and **empid** number of each employee and of the employee's manager, and the path within the hierarchy to **Henry**. The CONNECT BY clause uses the equality predicate `PRIOR empid = mgrid` to return the employees who report to the managers (in this case, only**Henry**) whose **empid** was returned by the previous step. The result set of the query is:

```
empid      12
name       Henry
mgrid      16
hierarchy  /Henry

empid      9
name       Shoeman
mgrid      12
hierarchy  /Henry/Shoeman

empid      8
name       Smith
mgrid      12
hierarchy  /Henry/Smith

empid      7
name       O'Neil
mgrid      12
hierarchy  /Henry/O'Neil

4 row(s) retrieved.
```

These rows are listed in the order in which they were retrieved:

- The START WITH clause returned the **Henry** row at the root of this hierarchy.
- The first step of the CONNECT BY clause returned three rows, corresponding to the three employees who report to **Henry**.
- The next CONNECT BY step returned no rows, because the **Shoeman**, **Smith**, and **O'Neil** rows that returned by the previous step are all leaf nodes within this hierarchy, for which the `PRIOR empid = mgrid` condition evaluates to false.

Query execution ended, returning the four rows that are shown, where **hierarchy** is an alias for the path to **Henry** that SYS_CONNECT_BY_PATH(`name,'/'`) returned for each row. (In the first returned row, the string `/Henry` shows the root status of **Henry**.)

Dependency patterns that are not a simple graph

You can run a recursive hierarchical query on a data set that includes complex dependencies, such as multiple root nodes, or multiple parent nodes for child nodes. However, cycling might occur and more records might be returned, based on hierarchical data tree structures.

The CONNECT BY clause cannot analyze data sets that include cycles where some child nodes are identified as ancestors of their parent node. If your data set includes cycles, the cycles might be artifacts of rows that contain invalid data.

If the data set that the self-referential table describes includes more than one hierarchy, include the START WITH clause with a condition that is only true for the root of the hierarchy that you want the recursive query to return. Run a separate query on the table for each hierarchy, using a different START WITH clause to specify the root in each query.

GROUP BY Clause

Use the GROUP BY clause to produce a single row of results for each group. A *group* is a set of rows that have the same values for each column (or column expression) that is referenced in this clause.

This syntax fragment is part of the SELECT statement on page 690.

(explicit id unique_989_Connect_42_gbcl002) unique_989_Connect_42_gbcl002

GROUP BY Clause

" GROUP BY "

" { | [*table_object* .] *column* | [] { *col_alias* | *select_number* } } "

Element	Description	Restrictions	Syntax
col_alias	Alias for a column name	Must have been declared in the Projection clause	Identifier on page 1231
column	Group rows by the value of this column (or of this expression)	See Dependencies between the GROUP BY and Projection clauses on page 753.	Identifier on page 1231, Expression on page 1003
select_number	Integer specifying the ordinal position of a column or expression in the select list of the Projection clause	See the Select numbers section.	Literal Number on page 1198
table_object	Name, synonym, or alias of the table or view containing *column*	Must exist and must be specified in the FROM clause	Identifier on page 1231

The SELECT statement with a GROUP BY clause returns a single row of results for each group of rows that have the same value in *column*, or that have the same value in the column that *col_alias* references, or that have the same value in the column or expression that the *select_number* specifies.

In an NLSCASE INSENSITIVE database, collation and string comparisons on NCHAR and NVARCHAR data disregard lettercase differences, so that the database server treats case variants among strings composed of same sequence letters as duplicates. For queries that group data on NCHAR or NVARCHAR columns, if some of the qualifying rows differ only in letter case, the number of groups will be smaller than from the same query on the same data set in a case-sensitive database. For more information on data processing in databases that were created with the NLSCASE INSENSITIVE property, see Duplicate rows in NLSCASE INSENSITIVE databases on page 699 and NCHAR and NVARCHAR expressions in case-insensitive databases.

In a column listed in a GROUP BY clause, each row that contains a NULL value belongs to a single group. That is, all NULL values are grouped together.

Select numbers

You can use one or more integers in the GROUP BY clause to stand for column expressions. In the next example, the first SELECT statement uses select numbers for **order_date** and **paid_date - order_date** in the GROUP BY clause. You can group only by a combined expression using the select numbers.

In the second SELECT statement, you cannot replace the `2` with the arithmetic expression **paid_date - order_date**:

```
SELECT order_date, COUNT(*), paid_date - order_date
   FROM orders GROUP BY 1, 3;
SELECT order_date, paid_date - order_date
   FROM orders GROUP BY order_date, 2;
```

Dependencies between the GROUP BY and Projection clauses

The GROUP BY clause restricts what the Projection clause can specify. If you include a GROUP BY clause, each *column* in the select list of the Projection clause must also be referenced in the GROUP BY clause. Some column data types and column expressions that are valid in the Projection clause, however, cannot be referenced in the GROUP BY clause.

If a column expression in the select list is only the column name or alias, you must use its name or its alias in the GROUP BY clause, if none of the restrictions below apply to the column. If a column is combined with another column by an arithmetic operator, you can choose to group the query result set in either of two ways:

- By the names or aliases of the individual columns,
- or else by the combined expression using the *select number*, a literal integer that specifies the ordinal position of that expression within the select list of the Projection clause.

The following restrictions, however, prevent some columns and expressions from being included in the GROUP BY clause, or can prevent the query from including the GROUP BY clause.

- Constant expressions are not valid in the GROUP BY clause.
- You cannot include a ROWID in a GROUP BY clause.
- If the Projection clause includes a BYTE or TEXT column or a BYTE or TEXT column expression, the query cannot include the GROUP BY clause.
- If the Projection clause includes a column of a user-defined data type, the column cannot be used in a GROUP BY clause unless the UDT can use the built-in bit-hashing function. Any UDT that cannot use the built-in bit-hashing function must be created with the CANNOTHASH modifier, which tells the database server that the UDT cannot be used in a GROUP BY clause.

The following section identifies restrictions on the GROUP BY clause when the Projection clause includes aggregate expressions or time expressions.

Columns in aggregate expressions and time expressions

If you specify an aggregate function and one or more column expressions in the select list of a query that includes the GROUP BY clause, the GROUP BY clause must list the name or the alias of each column in the select list that is not used as part of an aggregate or of a time expression. The query fails with error -321, however, if the GROUP BY clause includes a select number corresponding to the ordinal position of an aggregate expression within the select list.

If an OLAP window function is specified in the Projection clause of a query that includes the GROUP BY clause, all column references within the OLAP window function must also be included in the GROUP BY clause. The database server issues an error, however, if any of the columns referenced in the GROUP BY clause are operands of an aggregate expression or of a time expression.

The GROUP BY clause in the following example specifies one column that is not in an aggregate expression. Here the **total_price** column must not be in the GROUP BY list, because it appears in the Projection clause as the argument to an aggregate function.

```
SELECT order_num, COUNT(*), SUM(total_price)
   FROM items GROUP BY order_num;
```

During execution of this query, the COUNT and SUM aggregates are applied to each **order_num** group, rather than calculating the number of orders and the sum of their **total_price** values across the set of all rows in the **items** table.

HAVING Clause

Use the HAVING clause to apply one or more qualifying conditions to groups or to the entire result set.

This syntax fragment is part of the SELECT statement on page 690.

(explicit id unique_990_Connect_42_hacl001) unique_990_Connect_42_hacl001

HAVING Clause

" **HAVING** "

" <Condition> [] "

In the following examples, each condition compares one calculated property of the group with another calculated property of the group or with a constant. The first SELECT statement uses a HAVING clause that compares the calculated expression `COUNT(*)` with the constant `2`. The query returns the average total price per item on all orders that have more than two items.

The second SELECT statement lists customers and the call months for customers who have made two or more calls in the same month:

```
SELECT order_num, AVG(total_price) FROM items
   GROUP BY order_num HAVING COUNT(*) > 2;
SELECT customer_num, EXTEND (call_dtime, MONTH TO MONTH)
   FROM cust_calls GROUP BY 1, 2 HAVING COUNT(*) > 1;
```

You can use the HAVING clause to place conditions on the GROUP BY column values as well as on calculated values. This example returns **cust_code** and **customer_num**, **call_dtime**, and groups them by **call_code** for all calls that have been received from customers with **customer_num** less than 120:

```
SELECT customer_num, EXTEND (call_dtime), call_code
   FROM cust_calls GROUP BY call_code, 2, 1
   HAVING customer_num < 120;
```

The HAVING clause generally complements a GROUP BY clause. If you omit the GROUP BY clause, the HAVING clause applies to all rows that satisfy the query, and all rows in the table make up a single group. The following example returns the average price of all the values in the table, as long as more than ten rows are in the table:

```
SELECT AVG(total_price) FROM items HAVING COUNT(*) > 10;
```

Because conditions in the WHERE clause cannot include aggregate expressions, you can use the HAVING clause to apply conditions with aggregates to the entire result set of a query, as in the example above.

The condition in the HAVING clause cannot include a DISTINCT or UNIQUE aggregate expression. For example, the following query fails with a syntax error:

```
SELECT order_num, COUNT(*) number, AVG (total_price) average
   FROM items
   GROUP BY order_num
      HAVING COUNT(DISTINCT *) > 2;
```

No error is issued, however, if the DISTINCT keyword is omitted from the example above.

Set operators in combined queries

The set operators UNION, UNION ALL, INTERSECT, and MINUS can manipulate the result sets of two queries that specify the same number of columns in the Projection clause, and that have compatible data types in the corresponding columns of both queries.

(The MINUS set operator has EXCEPT as its keyword synonym. Results that the MINUS and EXCEPT operators return from the same operands are always identical.)

These operators perform basic set operations of *union*, *intersection*, and *difference* on the result sets of two queries that are the left and right operands of the set operators:

- The UNION set operator combines the qualifying rows from two queries into a single result set that consists of the distinct rows that either or both of the queries returned. (If you also include the ALL keyword, the UNION ALL result set can include duplicate rows.)
- The INTERSECT set operator compares the result sets from two queries, but returns only the distinct rows that are in the result sets of both queries.
- The MINUS set operator compares the result sets from two queries, but returns only the distinct rows in the result set of the left query that are absent from the result set of the right query.

The set operators are useful in business analytic contexts. They can also be used in SELECT statements to check the integrity of your database after you have performed a DML operation like UPDATE, INSERT, DELETE, or MERGE. The set operators can similarly be used when you transfer data to a history table, for example, when you need to verify that the correct data is in the history table before you delete rows from the original table.

All of the set operators have the same precedence. In complex queries that include more than one set operator, the precedence of operators is from left to right. Use parentheses to group set operators and their operands, if you need to override the default left-to-right precedence of set operators.

Only the UNION set operator supports the ALL keyword. The ALL keyword is not valid with the INTERSECT, MINUS, or EXCEPT set operators, from which only distinct rows are returned.

When comparing rows to calculate a set intersection or difference, two NULL values are considered equal in INTERSECT and MINUS operations.

Restrictions on a Combined SELECT

Several restrictions apply to queries that you can combine with the UNION, INTERSECT, MINUS, or EXCEPT set operators.

- The number of items in the Projection clause of each query must be the same, and the corresponding items in each Projection clause must have compatible data types.
- The Projection clause of each query cannot specify BYTE or TEXT objects. (This restriction does not apply to UNION ALL operations.)
- If a combined query includes the ORDER BY clause of the SELECT statement, it must follow the last Projection clause, and you must specify each ordered item by its integer *select_number*, not by its SQL identifier. Sorting takes place after the set operation is complete.

- You can store the combined results of any set operator in a temporary table, but the INTO TEMP clause can appear only in the final SELECT statement.
- In , you cannot use an INTO clause in a compound query unless exactly one row is returned, and you are not using a cursor. In this case, the INTO clause must be in the first SELECT statement that the set operator combines.

A *UNION subquery* is a query that includes the UNION or UNION ALL operator within a subquery. The following additional restrictions affect UNION subqueries, but they do not apply to combined queries that include the INTERSECT, MINUS, or EXCEPT set operators:

- The CREATE VIEW statement cannot specify a UNION subquery to define the view.
- Only columns in the local database are valid in a UNION subquery. You cannot reference a remote table or view in a UNION subquery.

Unlike UNION, the INTERSECT, MINUS, and EXCEPT set operators are valid in the following contexts:

- In combined queries that reference columns of tables in other databases of the local Informix® server instance, and in tables of other Informix® server instances.
- In view definitions. (You cannot, however, specify WITH CHECK OPTION in a CREATE VIEW statement that also includes a set operator.)

The following restrictions, however, affect all combined queries, including UNION and UNION ALL subqueries and queries that include the INTERSECT, MINUS, or EXCEPT set operators:

- Combined queries cannot be triggering events. If a valid combined query specifies a column on which a Select trigger has been defined, the query succeeds, but the trigger (or the INSTEAD OF trigger on a view) is ignored.
- General expressions that include host variables are not valid on the left of the ALL, ANY, IN, NOT IN and SOME operators in a query that includes a UNION subquery or any other set operator. An expression that consists solely of a single host variable, however, is valid in this context.

For example, the following query is valid under the above restriction:

```
SELECT col1 FROM tab1 WHERE ?  <= ALL
   (SELECT col2 FROM tab2 UNION SELECT col3 FROM tab3);
```

In this example, the expression to the left of ALL is a single host variable (?), which is the only expression involving host variables that is supported before the ALL, ANY, IN, NOT IN, or SOME operators in a query that also includes a set operator.

In contrast, the following example shows an invalid query:

```
SELECT col1 FROM tab1 WHERE (? + 8) <= ALL
   (SELECT col2 FROM tab2 UNION SELECT col3 FROM tab3);
```

This query fails because an operand of the `<=` relational operator to the left of the ALL operator is `(? + 8)`. An arithmetic expression that includes a host variable is not valid syntax in a UNION subquery, nor in queries that are combined by any other set operator.

Expressions that do not contain host variables are not subject to this restriction. Thus, the following query (that includes the same UNION subquery) is valid:

```
SELECT col1 FROM tab1 WHERE (col1 + 8) <= ALL
   (SELECT col2 FROM tab2 UNION SELECT col3 FROM tab3);
```

UNION Operator

Place the UNION operator between two SELECT statements to combine the queries into a single query.

You can string several SELECT statements together using the UNION operator. Corresponding items do not need to have the same name. Omitting the ALL keyword excludes duplicate rows.

UNION ALL operator

If you use the UNION ALL operator, all the qualifying rows from both queries are returned, without excluding any duplicate rows. (If you combine two queries by using the UNION operator without the ALL keyword, any duplicate rows are removed from the combined set of qualifying rows. That is, if multiple rows contain identical values in the corresponding columns or expressions that the Projection clauses of both queries specify, only one row from each set of duplicates is retained in the result set.)

The next example uses UNION ALL to combine the results of two SELECT statements without removing duplicates. The query returns a list of all the calls that were received during the first quarter of 2007 and the first quarter of 2008.

```
SELECT customer_num, call_code FROM cust_calls
   WHERE call_dtime BETWEEN
          DATETIME (2007-1-1) YEAR TO DAY
      AND DATETIME (2007-3-31) YEAR TO DAY
UNION ALL
SELECT customer_num, call_code FROM cust_calls
   WHERE call_dtime BETWEEN
          DATETIME (2008-1-1)YEAR TO DAY
      AND DATETIME (2008-3-31) YEAR TO DAY;
```

If you want to remove duplicates from the result set, use UNION without the keyword ALL as the set operator between the queries. In the preceding example, if the combination `101 B` were returned by both SELECT statements, the UNION operator would cause the combination to be listed only once. (If you want to remove duplicates within each SELECT statement, use the DISTINCT or UNIQUE keyword immediately before the Select list of the Projection clause, as described in Controlling duplicate return values on page 698.)

The ALL keyword is valid for specifying set operations only with the UNION operator. The database server issues an error if ALL immediately follows the INTERSECT, MINUS, or EXCEPT set operators, which exclude duplicates.

For information on how the database server identifies duplicate NCHAR and NVARCHAR values in databases that have the NLCASE INSENSITIVE property, see the topic NCHAR and NVARCHAR expressions in case-insensitive databases.

UNION in subqueries

You can use the UNION and UNION ALL operators in subqueries of SELECT statements within the WHERE clause, the FROM clause, and in collection subqueries. In this release of Informix®, however, subqueries that include UNION or UNION ALL are not supported in the following contexts:

- In the definition of a view
- In the event or in the Action clause of a trigger
- With the FOR UPDATE clause or with an Update cursor
- In a distributed query (accessing tables outside the local database)

For more information about collection subqueries, see Collection Subquery on page 965. For more information about the FOR UPDATE clause, see FOR UPDATE Clause on page 771.

In a combined subquery, the database server can resolve a column name only within the scope of its qualifying table reference. The following query, for example, returns an error:

```
SELECT * FROM t1 WHERE EXISTS
   (SELECT a FROM t2
   UNION
   SELECT b FROM t3 WHERE t3.c IN
      (SELECT t4.x FROM t4 WHERE t4.4 = t2.z));
```

Here **t2.z** in the innermost subquery cannot be resolved, because **z** occurs outside the scope of reference of the table reference **t2**. Only column references that belong to **t4**, **t3**, or **t1** can be resolved in the innermost subquery. The scope of a table reference extends downwards through subqueries, but not across the UNION operator to sibling SELECT statements.

INTERSECT Operator

When two queries are combined by this set operator, INTERSECT calculates the set intersection of the rows returned by the two queries that are its operands.

The rows that INTERSECT returns are present in the results sets of both the left and the right SELECT statements. The INTERSECT results are always distinct or unique rows, because INTERSECT eliminates any duplicate rows.

Consider the following example, where the table **t1** has following rows:

```
create table t1 (col1 int);
insert into t1 values (1);
insert into t1 values (2);
insert into t1 values (2);
insert into t1 values (2);
insert into t1 values (3);
insert into t1 values (4);
insert into t1 values (4);
insert into t1 values (NULL);
insert into t1 values (NULL);
insert into t1 values (NULL);
```

In the same example, table **t2** has these rows:

```
create table t2 (col1 int);
insert into t2 values (1);
insert into t2 values (3);
insert into t2 values (4);
insert into t2 values (4);
insert into t2 values (NULL);
```

The following query returns the distinct rows from both the query on the left and right sides of the INTERSECT operand. The important thing to be noted here is the result has a NULL value. Because the NULL value in table **t2** is considered to be equal when table **t2** is compared to table **t1**, so a NULL value from the set intersection is returned in the combined result set:

```
SELECT col1 FROM t1 INTERSECT SELECT col1 FROM t2;

       col1


          1
          3
          4
4 row(s) retrieved.
```

The INTERSECT operator has some (but not all) of the same restrictions as the UNION operator, but INTERSECT does not support the ALL keyword that enables UNION to return duplicate values. See also the topic Restrictions on a Combined SELECT on page 756.

MINUS operator

When two queries are combined by this set operator, the MINUS operator calculates the set difference between the rows returned by the SELECT statement on the left side and the rows returned by the SELECT statement on the right side.

The MINUS operator returns only the rows that are present in first result set but that are not in the second set. The MINUS results are always distinct or unique rows, because MINUS eliminates any duplicate rows.

For the same data set that is listed in the INTERSECT Operator on page 759 topic, the following query returns all the distinct rows from result set of the query to the left of the MINUS operator that are not in the result set of the query on the right:

```
SELECT col1 FROM t1 MINUS SELECT col1 FROM t2;

       col1

          2

1 row(s) retrieved.
```

The MINUS operator has some (but not all) of the same restrictions as the UNION operator, but MINUS does not support the ALL keyword that enables UNION to return duplicate values. See also the topic Restrictions on a Combined SELECT on page 756.

ORDER BY Clause

The ORDER BY clause sorts query results by specified columns or expressions.

This syntax fragment is part of the SELECT statement on page 690.

(explicit id unique_991_Connect_42_havcl002) unique_991_Connect_42_havcl002(explicit id unique_991_Connect_42_havcl001)
unique_991_Connect_42_havcl001(explicit id unique_991_Connect_42_havcl003) unique_991_Connect_42_havcl003(explicit id
unique_991_Connect_42_havcl005) unique_991_Connect_42_havcl005(explicit id unique_991_Connect_42_havcl006) unique_991_Connect_42_havcl006(explicit
id unique_991_Connect_42_havcl007) unique_991_Connect_42_havcl007(explicit id unique_991_Connect_42_havcl008)
unique_991_Connect_42_havcl008(explicit id unique_991_Connect_42_havcl009) unique_991_Connect_42_havcl009

ORDER BY Clause

" **ORDER [SIBLINGS]** [] **BY** "

" { { [*table* .] *column* | <Expression> [] } [<Substring>] | { *select_number* | *display_label* | <CASE expression> [] |
[] **ROWID** | [] <OLAP window function> [] } } [{ **ASC** | [**DESC**] }] [**NULLS** { **FIRST** | **LAST** }] [] "

Substring

" [] [*first, last*] [] "

Element	Description	Restrictions	Syntax
column	Sort rows by value in this column	None	Identifier on page 1231
display_label	Temporary name for a column or for a column expression	Must be unique among labels declared in the Projection clause	Identifier on page 1231
first, last	First and last byte in column substring to sort the result set	Integers; for BYTE, TEXT, and character data types only	Literal Number on page 1198
select_number	Ordinal position of a column in the select list of the Projection clause	See GROUP BY Clause on page 752.	Literal Number on page 1198
table	Name, synonym, or alias of the table or view containing *column*	Must exist and must be specified in the FROM clause	Identifier on page 1231

The ORDER BY clause implies that the query returns more than one row. In SPL, the database server issues an error if you specify the ORDER BY clause without a FOREACH loop to process the returned rows individually within the SPL routine.

The following query specifies a derived table in the FROM clause whose rows are ordered by the **col1** value, and declares **vtab** as the name of the derived table, and **vcol** as the name of its only column:

```
SELECT vcol FROM
   (SELECT FIRST 5 col1 FROM tab1 ORDER BY col1) vtab(vcol);
```

ORDER BY in NLSCASE INSENSITIVE databases

In databases created with the NLSCASE INSENSITIVE property, operations on columns and expressions of NCHAR or NVARCHAR data types make no distinction between upper case and lower case letters. For this reason, queries that include the ORDER BY clause might return rows in a sequence that disregard variants in letter case, if the column or expression are of NLS data types, and the data includes values that differ only in letter case.

If the data set includes letter case variants of the same string, these will be processed as duplicates, with case variants listed in their order of retrieval. For example, a set of NCHAR or NVARCHAR strings that were processed as duplicates might appear in this order:

```
gAMma
GAmma
GaMMa
gamma
GAMMA
```

For more information, see Duplicate rows in NLSCASE INSENSITIVE databases on page 699 and NCHAR and NVARCHAR expressions in case-insensitive databases.

Ordering by a Column or by an Expression

To order query results by an expression, you must also declare a display label for the expression in the Projection clause, as in the following example, which declares the display label **span** for the difference between two columns:

```
SELECT paid_date - ship_date span, customer_num FROM orders
   ORDER BY span;
```

Informix® supports columns and expressions in the ORDER BY clause that do not appear in the select list of the Projection clause. You can omit a display label for the derived column in the select list and specify the derived column by means of a select number in the ORDER BY clause.

The select list of the Projection clause must include any column or expression that the ORDER BY clause specifies, however, if any of the following is true:

- The query includes the DISTINCT, UNIQUE, or UNION operator.
- The query includes the INTO TEMP *table* clause.
- The distributed query accesses a remote database whose server requires every column or expression in the ORDER BY clause to also appear in the select list of the Projection clause.
- An expression in the ORDER BY clause includes a display label for a column substring. (See the next section, Ordering by a Substring on page 763.)

The next query selects one column from the **orders** table and sorts the results by the value of another column. By default, the rows are listed in ascending order.

```
SELECT ship_date FROM orders ORDER BY order_date;
```

You can order by an aggregate expression only if the query also has a GROUP BY clause. This query declares the display label **maxwgt** for an aggregate in the ORDER BY clause:

```
SELECT ship_charge, MAX(ship_weight) maxwgt
   FROM orders GROUP BY ship_charge ORDER BY maxwgt;
```

If the current processing locale defines a localized collation, then NCHAR and NVARCHAR column values are sorted in that localized order.

In Informix®, no *column* in the ORDER BY clause can be a collection type, but a query whose result set defines a collection-derived table can include the ORDER BY clause. For an example, see Collection-Derived Table on page 1212.

You might improve the performance of some non-PDQ queries that use the ORDER BY clause to sort a large set of rows if you increase the setting of the DS_NONPDQ_QUERY_MEM configuration parameter.

Ordering by a Substring

You can order by a substring instead of by the entire length of a character, BYTE, or TEXT column, or of an expression returning a character string. The database server uses the substring to sort the result set. Define the substring by specifying integer subscripts (the *first* and *last* parameters), representing the starting and ending byte positions of the substring within the column value.

The following SELECT statement queries the **customer** table and specifies a column substring in the ORDER BY column. This instructs the database server to sort the query results by the portion of the **lname** column contained in the sixth through ninth bytes of the column value.

```
SELECT * from customer ORDER BY lname[6,9];
```

Assume that the value of **lname** in one row of the **customer** table is `Greenburg`. Because of the column substring in the ORDER BY clause, the database server determines the sort position of this row by using the value `burg`, rather than the complete column value `Greenburg`.

When ordering by an expression, you can specify substrings only for expressions that return a character data type. If you specify a column substring in the ORDER BY clause, the column must have one of the following data types: BYTE, CHAR, NCHAR, NVARCHAR, TEXT, or VARCHAR.

Informix® can also support LVARCHAR column substrings in the ORDER BY clause, if the column is in a database of the local database server.

For information on the GLS aspects of using column substrings in the ORDER BY clause, see the *HCL® Informix® GLS User's Guide*.

Ordering by a CASE expression

The ORDER BY clause can include CASE expressions to specify a sorting key.

In the following example, column a_col of table tab_case is of type INT. The query on table tab_case includes both column a_col and the aggregate expression SUM(a_col) in the Projection list, and groups the results by the value of a_col. The ORDER BY clause specifies two sorting keys:

- A CASE expression that immediately follows the ORDER BY keywords
- The AVG(a_col) aggregate expression:

```
CREATE TABLE tab_case(a_col INT, b_col VARCHAR(32));

SELECT  a_col, SUM(a_col)
   FROM tab_case
   GROUP BY a_col
   ORDER BY CASE
               WHEN a_col IS NULL
                  THEN 1
               ELSE 0 END ASC,
            AVG(a_col);
```

Here the ASC keyword explicitly identifies the result of the CASE expression as an ascending sort key. By default, the AVG(a_col) sorting key also specifies an ascending order.

In the following similar example, based on a query on the same tab_case table, a second CASE expression returns either 1 or 0 as the sorting key value for the returned AVG(a_col) aggregate values.

```
SELECT  a_col, SUM(a_col)
   FROM tab_case GROUP BY a_col
   ORDER BY CASE
               WHEN a_col IS NULL
                  THEN 1
               ELSE 0 END ASC,
            AVG(a_col),
            CASE
               WHEN AVG(a_col) IS NULL
                  THEN 1
               ELSE 0 END;
```

Ascending and Descending Orders

You can use the ASC and DESC keywords to specify ascending (smallest value first) or descending (largest value first) order.

The default order is ascending. For DATE and DATETIME data types, *smallest* means earliest in time and *largest* means latest in time. For character data types in the default locale, the order is the ASCII collating sequence, as listed in #unique_1071.

For NCHAR or NVARCHAR data types, the localized collating order of the current session is used, if that is different from the code set order. For more information about collation, see SET COLLATION statement on page 775.

If you specify the ORDER BY clause, NULL values by default are ordered as less than values that are not NULL. Using the ASC order, a NULL value comes before any non-NULL value; using DESC order, the NULL comes last.

Specifying the order of NULL values

The ORDER BY clause can include the NULLS FIRST keywords or the NULLS LAST keywords to show explicitly (or else to override) the default sort order of NULL values:

- The NULLS FIRST keywords instruct the database server to put NULL values first in the sorted query results. In an ascending sort, the ASC NULLS FIRST keywords request the default order. In a descending sort, DESC NULLS FIRST specifies that rows with a NULL value in the sort key column precede non-NULL rows in the sorted result set.
- The NULLS LAST keywords instruct the database server to put NULL values last in the sorted query results. In a descending sort, the DESC NULLS LAST keywords request the default order. In an ascending sort, ASC NULLS LAST specifies that rows with a NULL value in the sort key column follow non-NULL rows in the sorted result set.

Nested Ordering

If you list more than one column in the ORDER BY clause, your query is ordered by a nested sort. The first level of sort is based on the first column; the second column determines the second level of sort. The following example of a nested sort selects all the rows in the **cust_calls** table and orders them by **call_code** and by **call_dtime** within **call_code**:

```
SELECT * FROM cust_calls ORDER BY call_code, call_dtime;
```

Using Select Numbers

In place of column names, you can enter in the ORDER BY clause one or more integers that refer to the position of items listed in the select list of the Projection clause. You can also use a select number to sort by an expression.

The following example orders by the expression **paid_date - order_date** and **customer_num**, using select numbers in a nested sort:

```
SELECT order_num, customer_num, paid_date - order_date
   FROM orders
   ORDER BY 3, 2;
```

Select numbers are required in the ORDER BY clause when SELECT statements are joined by the UNION or UNION ALL keywords, or when compatible columns in the same position have different names.

Ordering by Rowids

You can specify the ROWID keyword in the ORDER BY clause. This specifies the **rowid** column, a hidden column in nonfragmented tables and in fragmented tables that were created with the WITH ROWIDS clause. The **rowid** column contains a unique internal record number that is associated with a row in a table. (It is recommended, however, that you utilize primary keys as your access method, rather than exploiting the **rowid** column.)

The ORDER BY clause cannot specify the **rowid** column if the table from which you are selecting is a fragmented table that has no **rowid** column.

You do not need to include the ROWID keyword in the Projection clause when you specify ROWID in the ORDER BY clause.

For further information about **rowid** values and how to use the **rowid** column in column expressions, see WITH ROWIDS Option on page 41 and Using Rowids on page 1031.

ORDER BY Clause with DECLARE

In , you cannot use a DECLARE statement with a FOR UPDATE clause to associate a cursor with a SELECT statement that has an ORDER BY clause.

Placing Indexes on ORDER BY Columns

When you include an ORDER BY clause in a SELECT statement, you can improve the performance of the query by creating an index on the column or columns that the ORDER BY clause specifies. The database server uses the index that you placed on the ORDER BY columns to sort the query results in the most efficient manner. For more information on how to create indexes that correspond to the columns of an ORDER BY clause, see Using the ASC and DESC Sort-Order Options on page 246.

ORDER SIBLINGS BY Clause

The ORDER SIBLINGS BY clause is valid only in a hierarchical query. The optional SIBLINGS keyword specifies an order that first sorts the parent rows, and then sorts the child rows of each parent for every level within the hierarchy.

Rows that have duplicate lists of values in the columns specified after the SIBLINGS BY keywords are arbitrarily ordered among the rows with the same list of values and the same parent. If a hierarchical query includes the ORDER BY clause without the SIBLINGS keyword, rows are ordered according to the sort specifications that follow the ORDER BY keywords. Neither the ORDER BY clause nor the ORDER SIBLINGS BY option to the ORDER BY clause is required in hierarchical queries.

The hierarchical query in the following example returns the subset of rows in the hierarchical data set whose root is **Goyal**, as listed in the topic Hierarchical Clause on page 738. This query includes the ORDER SIBLINGS BY clause to sort by **name** the employees who report to the same manager:

```
SELECT empid, name, mgrid, LEVEL
   FROM employee
      START WITH name = 'Goyal'
      CONNECT BY PRIOR empid = mgrid
   ORDER SIBLINGS BY name;
```

The rows returned by this query are sorted in the following order:

```
         empid name            mgrid       level

         16 Goyal              17           1
         12 Henry              16           2
          7 O'Neil             12           3
          9 Shoeman            12           3
          8 Smith              12           3
         14 Scott              16           2
         11 Zander             16           2
          6 Barnes             11           3
          5 McKeough           11           3

9 row(s) retrieved.
```

Here the START WITH clause returned the **Goyal** row at the root of this hierarchy. Two subsequent CONNECT BY steps (marked as `2` and `3` in the **level** pseudocolumn) returned three sets of sibling rows:

- **Henry**, **Scott**, and **Zander** are siblings whose parent is **Goyal**;
- **O'Neil**, **Shoeman**, and **Smith** are siblings whose parent is **Henry**;
- **Barnes** and **McKeough** are siblings whose parent is **Zander**.

The next CONNECT BY step returned no rows, because the rows for which **level** = `3` are leaf nodes within this hierarchy. At this point in the execution of the query, the ORDER SIBLINGS BY clause was applied to the result set, sorting the rows in the order shown above.

Because the sort key, **name**, is a VARCHAR column, the returned rows within each set of siblings are in the ASCII order of their **employee.name** values. Only the sets of siblings that are leaf nodes in the hierarchy of returned rows appear consecutively in the sorted result set, because the managers are immediately followed by the employees who report to them, rather than by their siblings. An exception in this example is **Scott**, whose child nodes form an empty set.

The SIBLINGS keyword in the ORDER BY clause is an extension to the ISO standard syntax for the SQL language. The SELECT statement fails with an error if you include the SIBLINGS keyword in the ORDER BY clause of a query or subquery that does not include a valid CONNECT BY clause.

For more information about hierarchical queries and the CONNECT BY clause, see Hierarchical Clause on page 738.

OLAP window functions in the ORDER BY clause of SELECT statements

You can include OLAP window functions in the final ORDER BY clause of SELECT statements that do not include the CONNECT BY clause.

If an OLAP function clause is present in the ORDER BY clause, then the OLAP function is evaluated first, before the ORDER BY evaluation.

More generally, for simple SELECT statements that includes one or more OLAP window functions, the database server follows this sequence of processing:

- Apply any joins, filters, GROUP BY, or HAVING specifications to obtain the set of qualifying rows to return as the query result.
- Create window partitions of qualifying rows and apply the specified OLAP functions to each row in each partition (or to the entire query result set, if no partitions are defined).
- Apply the ORDER BY clause of the SELECT statement to the final query result.

For nested queries, each subquery follows the order above, but OLAP window partitions and their OLAP functions are applied to the result set of the innermost subquery in which the OLAP window is defined.

If the OLAP window includes a window ORDER clause, that clause, rather than the ORDER BY clause of the SELECT statement, defines the row numbers that the window ROW_NUMBER function assigns to the rows in partitions of the same OLAP window. The window ORDER clause does not, however, define the ordering of the query result set, which the ORDER BY clause of SELECT statement defines.

If the OLAP window does not include a window ORDER clause, the row numbers that the window ROW_NUMBER function assigns to the rows are in arbitrary order, as returned by the query or subquery, rather than according to any ORDER BY clause of the SELECT statement.

Ordering STANDARD or RAW result tables

When the SELECT INTO Table clause defines a permanent table to store the result of a query, any non-trivial column expression in that clause must also declare an alias for the corresponding column in the newly-created result table. To specify that column as a sorting key for the result table, the ORDER BY clause must also reference the same alias, rather than specifying the non-trivial column expression.

For example, in the following nested query, **tab56** is the identifier of a result table, and **tab56_col0** is an alias for the non-trivial column expression that a subquery in the Projection clause defines. The ORDER BY clause specifies the same subquery as a sorting key, rather than referencing that non-trivial column expression by its alias:

```
SELECT ( SELECT tab54.tab54_col7 tab56_col0
    FROM tab54
    WHERE (tab54.tab54_col7 = -1423023 )
    ) tab56_col0,
  "" tab56_col1
FROM tab57
WHERE tab57.tab57_col1 == -6296233
ORDER BY  (
    SELECT tab54.tab54_col7 tab56_col0
    FROM tab54
    WHERE (tab54.tab54_col7 = -1423023 )
    ) NULLS FIRST,2 NULLS FIRST
INTO tab56;
```

Specifying the non-trivial column expression in the ORDER BY clause is acceptable in a normal SELECT statement, but not in an ORDER BY clause that sorts a result table that the SELECT INTO Table clause created. In the example above, the database server returns SQL error -19828.

To avoid this error, the example above must be modified to remove the non-trivial column expression from the ORDER BY clause, replacing that expression with its alias:

```
SELECT ( SELECT tab54.tab54_col7 tab56_col0
    FROM tab54
    WHERE (tab54.tab54_col7 = -1423023 )
    ) tab56_col0,
  "" tab56_col1
FROM tab57
WHERE tab57.tab57_col1 == -6296233
ORDER BY
    tab56_col0     -- Substituted alias for column expression in result table)
NULLS FIRST,2 NULLS FIRST
INTO tab56;
```

LIMIT Clause

The LIMIT clause can restrict the result set of the query to some maximum number of rows. If this clause specifies a value smaller than the number of qualifying rows, the query returns only a subset of the rows that satisfy the selection criteria.

The LIMIT clause is an extension to the ISO/ANSI standard for the SQL language. This syntax fragment is part of the SELECT statement on page 690.

(explicit id unique_993_Connect_42_limcl004) unique_993_Connect_42_limcl004

LIMIT clause

" [] **LIMIT** { *max* | *max_var* } "

Element	Description	Restrictions	Syntax
max	Integer (> 0) specifying maximum number of rows to return	If *max* > number of qualifying rows then all matching rows are returned	Literal Number on page 1198
max_var	Host variable or local SPL variable storing the value of *max*	Same as *max*; valid in prepared objects and in SPL routines	Language dependent

Usage

The LIMIT clause specifies that the result set includes no more than *max* rows (or exactly *max* rows, if *max* is less than the number of qualifying rows). Any additional rows that satisfy the query selection criteria are not returned.

The following example sets *max* with a literal integer to retrieve at most 10 rows from table **tab1**:

```
SELECT a, b FROM tab1 LIMIT 10;
```

You can also use a host variable, or the value of an SPL input parameter in a local variable, to assign the value of *max*.

If the LIMIT clause follows the ORDER BY clause, the returned rows are sorted according to the ORDER BY specifications. Because query results are generally not retrieved in any particular sequence, using ORDER BY to constrain the order of rows can be useful in queries whose LIMIT clause returns only a subset of the qualifying rows.

For example, the following query returns data about the ten highest-paid employees:

```
SELECT name, salary FROM emp
   ORDER BY salary DESC LIMIT 10;
```

You can use the LIMIT clause in a query whose result set defines a collection-derived table (CDT) within the FROM clause of another SELECT statement. The following query specifies a CDT that has no more than ten rows:

```
SELECT *
   FROM TABLE(MULTISET(SELECT * FROM employees
   ORDER BY employee_id LIMIT 10 )) vt(x,y), tab2
   WHERE tab2.id = vt.x;
```

The next example applies the LIMIT clause to the result of a UNION query:

```
SELECT a, b FROM tab1 LIMIT 10 UNION SELECT a, b FROM tab2;
```

Restrictions on the LIMIT clause

The LIMIT clause is not valid in any of the following contexts:

- In the definition of a view
- In nested SELECT statements
- In subqueries, except for subqueries that specify table expressions in the FROM clause
- In a singleton SELECT (where *max* = 1) within an SPL routine
- Where embedded SELECT statements are used as expressions.

Dependencies between the LIMIT clause and Projection clause

Syntax of the LIMIT clause that can follow the ORDER BY clause resembles the syntax of the FIRST option in the Projection clause, where LIMIT is valid as a keyword synonym for FIRST. The following examples are both valid, and both queries return the same results:

```
SELECT FIRST 5 c1, c2 FROM tab ORDER BY c3;
SELECT c1, c2 FROM tab ORDER BY c3 LIMIT 5;
```

Whether or not a query that uses the LIMIT clause also specifies ORDER BY can affect which qualifying rows are in the result sent if the Projection clause includes the SKIP options. The SKIP option, which specifies an offset for the *max* qualifying rows that FIRST or LIMIT restricts, is valid only in the Projection clause, but its specification can affect which of the sorted rows the LIMIT clause includes in the query result.

The following query retrieves two column values from each row of table tab, but the SKIP option excludes the first twenty rows. After ORDER BY sorts the rows by their value in a third column, the LIMIT clause restricts the query result to only the ten rows with the smallest values in column c3:

```
SELECT SKIP 20 c1, c2 FROM tab ORDER BY c3 LIMIT 10;
```

Combining the SKIP option and the LIMIT clause is also valid in queries that include table expressions in the FROM clause, as in this example:

```
SELECT * FROM (SELECT SKIP 2 col1 FROM tab1
                  WHERE col1 > 50 LIMIT 8);
```

But because you cannot use FIRST as a synonym for LIMIT outside the Projection clause, the following query fails with a syntax error:

```
SELECT SKIP 20 c1, c2 FROM tab ORDER BY c3 FIRST 10;
```

The database server also issues an error if the FIRST or LIMIT *max* value in the Projection clause differs from the *max* value in the LIMIT clause, if both *max* specifications are in the same query block and apply to the same query result set, as in these bad examples:

```
SELECT FIRST 20 c1, c2 FROM tab ORDER BY c3 LIMIT 10;
SELECT LIMIT 10 c1, c2 from tab ORDER BY c3 LIMIT 20;
```

No error is issued, however, if both the FIRST or LIMIT option in the Projection clause and the LIMIT clause specify the same *max* value. The following examples are equivalent and valid:

```
SELECT LIMIT 20 c1, c2 FROM tab ORDER BY c4;
SELECT c1, c2 FROM tab ORDER BY c4 LIMIT 20;
```

Besides the SKIP, FIRST, and LIMIT options, the DISTINCT and UNIQUE specifications of the Projection clause can also restrict query results to a subset of the qualifying rows, as their Projection clause topics explain.

FOR UPDATE Clause

Use the FOR UPDATE clause in ESQL/C applications and in DB-Access when you intend to update the values returned by a prepared SELECT statement when the values are fetched.

Preparing a SELECT statement that contains a FOR UPDATE clause is equivalent to preparing the SELECT statement without the FOR UPDATE clause and then declaring a FOR UPDATE cursor for the prepared statement. This syntax fragment is part of the SELECT statement on page 690.

FOR UPDATE Clause

(explicit id unique_995_Connect_42_fup006) unique_995_Connect_42_fup006

" [] **FOR UPDATE** "

" [**OF** *column*] "

Element	Description	Restrictions	Syntax
column	Name of a column that can be updated after a FETCH	Must be in the FROM clause *table*, but it need not be in the Projection list. All columns must be from the same table.	Identifier on page 1231

The FOR UPDATE keywords notify the database server that updating is possible, causing it to use more stringent locking than it would with a Select cursor. You cannot modify data through a cursor without this clause. You can specify which columns can be updated.

After you declare a cursor for a SELECT . . . FOR UPDATE statement, you can update or delete the currently selected row using an UPDATE or DELETE statement with the WHERE CURRENT OF clause. The keywords CURRENT OF refer to the row that was most recently fetched; they replace the usual conditional expressions in the WHERE clause. To update rows with a specific value, your program might contain statements such as those in the following example:

```
EXEC SQL BEGIN DECLARE SECTION;
    char fname[ 16];
    char lname[ 16];
    EXEC SQL END DECLARE SECTION;
. . .

 EXEC SQL connect to 'stores_demo';
 /* select statement being prepared contains a for update clause */
 EXEC SQL prepare x from 'select fname, lname from customer for update';
 EXEC SQL declare xc cursor for x;

 for (;;)
```

```
  {
  EXEC SQL fetch xc into $fname, $lname;
  if (strncmp(SQLSTATE, '00', 2) != 0) break;
  printf("%d %s %s\n",cnum, fname, lname );
  if (cnum == 999)              --update rows with 999 customer_num
      EXEC SQL update customer set fname = 'rosey' where current of xc;
  }

EXEC SQL close xc;
EXEC SQL disconnect current;
```

A SELECT . . . FOR UPDATE statement, like an Update cursor, allows you to perform updates that are not possible with the UPDATE statement alone, because both the decision to update and the values of the new data items can be based on the original contents of the row. The UPDATE statement cannot query the table that is being updated.

Note: A normal update inside the FETCH loop of a cursor cannot guarantee that the updated rows are not fetched again after the UPDATE. The WHERE CURRENT OF specification relates the UPDATE to the Update cursor, and guarantees that each row is updated no more than once, by internally keeping a list of the rows that have already been updated. These rows will not be fetched again by the Update cursor.

Restrictions

A SELECT statement that includes the FOR UPDATE clause must conform to the following restrictions:

- The statement can select data from only one table.
- The statement cannot include any aggregate functions.
- The statement cannot include any of the following clauses or keywords: DISTINCT, EXCEPT, FOR READ ONLY, GROUP BY, INTO TEMP, INTERSECT, INTO EXTERNAL, MINUS, ORDER BY, UNION, UNIQUE.
- DECLARE statements that associate a cursor with the statement cannot also include the FOR UPDATE keywords.
- The statement is valid only in ESQL/C routines and (within transactions) in the DB-Access utility. It cannot, for example, be issued within an SPL routine.

For information on how to declare an update cursor for a SELECT statement that does not include a FOR UPDATE clause, see Using the FOR UPDATE Option on page 438.

Update cursors in SPL routines

You cannot include the FOR UPDATE keywords in the SELECT . . . INTO segment of the FOREACH statement of SPL. An SPL routine, however, can provide the functionality of a FOR UPDATE cursor

- by declaring a *cursor* name in the FOREACH statement,
- and then using the WHERE CURRENT OF *cursor* clause in UPDATE or DELETE statements that operate on the current row of that *cursor* within the same FOREACH loop.

FOR READ ONLY Clause

Use the FOR READ ONLY keywords to specify that the Select cursor declared for the SELECT statement is a read-only cursor. A read-only cursor is a cursor that cannot modify data.

Normally, you do not need to include the FOR READ ONLY clause in a SELECT statement. SELECT is a read-only operation by definition, so the FOR READ ONLY clause is usually unnecessary. In certain circumstances, however, you must include the FOR READ ONLY keywords in a SELECT statement.

If you attempt to include both the FOR READ ONLY clause and the FOR UPDATE clause in the same SELECT statement, the SELECT statement fails. For information on declaring a read-only cursor for a SELECT statement that does not include a FOR READ ONLY clause, see DECLARE statement on page 433.

In an ANSI-compliant database, Select cursors are update cursors by default. An update cursor is a cursor that can be used to modify data. These update cursors are incompatible with the read-only mode of the database. For example, this SELECT statement against the **customer_ansi** table fails:

```
EXEC SQL declare ansi_curs cursor for
   select * from customer_ansi;
```

The solution is to include the FOR READ ONLY clause in your Select cursors. The read-only cursor that this clause specifies is compatible with the read-only mode of the database. For example, the following SELECT FOR READ ONLY statement against the **customer_ansi** table succeeds:

```
EXEC SQL declare ansi_read cursor for
   select * from customer_ansi for read only;
```

executes all SELECT statements with Select cursors, so you must specify FOR READ ONLY in all queries that access data in a read-only ANSI-compliant database. The FOR READ ONLY clause causes to declare the cursor for the SELECT statement as a read-only cursor.

For more information on level-0 backups, see your *HCL® Informix® Backup and Restore Guide*. For more information on Select cursors, read-only cursors, and update cursors, see DECLARE statement on page 433.

SET AUTOFREE statement

Use the SET AUTOFREE statement to instruct the database server to enable or disable a memory-management feature that can free the memory allocated for a cursor automatically, as soon as the cursor is closed.

Syntax

SET AUTOFREE { **ENABLED** | [{ **DISABLED** }] } [**FOR** { *cursor_id* | *cursor_id_var* }]

Element	Description	Restrictions	Syntax
cursor_id	Name of a cursor for which Autofree is to be reset	Must already be declared within the program	Identifier on page 1231

Element	Description	Restrictions	Syntax
cursor_id_var	Host variable that holds the value of *cursor_id*	Must store a cursor_id already declared in the program	Must conform to language-specific rules for names.

Usage

This statement is an extension to the ANSI/ISO standard for SQL. You can use this statement only with .

When the Autofree feature is enabled for a cursor, and the cursor is subsequently closed, you do not need to explicitly use the FREE statement to release the memory that the database server allocated for the cursor. If you issue SET AUTOFREE but specify no option, the default is ENABLED.

The SET AUTOFREE statement that enables the Autofree feature must appear before the OPEN statement that opens a cursor. The SET AUTOFREE statement does not affect the memory allocated to a cursor that is already open. After a cursor is Autofree enabled, you cannot open that cursor a second time.

Globally Affecting Cursors with SET AUTOFREE

If you include no FOR *cursor_id* or FOR *cursor_id_var* clause, then the scope of SET AUTOFREE is all subsequently-declared cursors in the program (or more precisely, all cursors declared before a subsequent SET AUTOFREE statement with no FOR clause globally resets the Autofree feature). This example enables the Autofree feature for all subsequent cursors in the program:

```
EXEC SQL set autofree;
```

The next example disables the Autofree feature for all subsequent cursors:

```
EXEC SQL set autofree disabled;
```

Using the FOR Clause to Specify a Specific Cursor

If you specify FOR *cursor _id* or FOR *cursor_id_var*, then SET AUTOFREE affects only the cursor that you specify after the FOR keyword.

This option allows you to override a global setting for all cursors. For example, if you issue a SET AUTOFREE ENABLED statement for all cursors in a program, you can issue a subsequent SET AUTOFREE DISABLED FOR statement to disable the Autofree feature for a specific cursor.

In the following example, the first statement enables the Autofree feature for all cursors, while the second statement disables the Autofree feature for the cursor named **x1**:

```
EXEC SQL set autofree enabled;
EXEC SQL set autofree disabled for x1;
```

Here the **x1** cursor must have been declared but not yet opened.

Associated and Detached Statements

When a cursor is automatically freed, its associated prepared statement (or associated statement) is also freed.

The term *associated statement* has a special meaning in the context of the Autofree feature. A cursor is associated with a prepared statement if it is the first cursor that you declare with the prepared statement, or if it is the first cursor that you declare with the statement after the statement is detached.

The term *detached statement* has a special meaning in the context of the Autofree feature. A prepared statement is detached if you do not declare a cursor with the statement, or if the cursor with which the statement is associated was freed.

If the Autofree feature is enabled for a cursor that has an associated prepared statement, and that cursor is closed, the database server frees the memory allocated to the prepared statement as well as the memory allocated for the cursor. Suppose that you enable the Autofree feature for the following cursor:

```
/*Cursor associated with a prepared statement */
EXEC SQL prepare sel_stmt 'select * from customer';
EXEC SQL declare sel_curs2 cursor for sel_stmt;
```

When the database server closes the **sel_curs2** cursor, it automatically performs the equivalent of the following FREE statements:

```
FREE sel_curs2;
FREE sel_stmt;
```

Because memory for the **sel_stmt** statement is freed automatically, you cannot declare a new cursor on it unless you prepare the statement again.

Closing Cursors Implicitly

A potential problem exists with cursors that have the Autofree feature enabled. In a database that is not ANSI-compliant, if you do not close a cursor explicitly and then open it again, the cursor is closed implicitly. This implicit closing of the cursor triggers the Autofree feature. The second time the cursor is opened, the database server generates an error message (`cursor not found`) because the cursor is already freed.

SET COLLATION statement

Use the SET COLLATION statement to specify a new collating order for the session, superseding the collation implied by the DB_LOCALE environment variable setting. SET NO COLLATION restores the default collation.

Syntax

SET { **COLLATION** *locale* | **NO COLLATION** }

Element	Description	Restrictions	Syntax
locale	Name of a locale whose collating order is to be used in this session	Must be the name of a locale that the database server can access	Quoted String on page 1202

Usage

The SET COLLATION statement is an extension to the ANSI/ISO standard for SQL. You can use this statement with .

As the *HCL® Informix® GLS User's Guide* explains, the database server uses locale files to specify the character set, the collating order, and other conventions of some natural language to display and manipulate character strings and other data values. The collating order of the database locale is the sequential order in which the database server sorts character strings.

If you set no value for DB_LOCALE, the default locale, based on United States English, is **en_us.8859-1** for UNIX™, or Code Page 1252 for Windows™ systems. Otherwise, the database server uses the DB_LOCALE setting as its locale. The SET COLLATION statement overrides the collating order of DB_LOCALE at runtime for all database servers previously accessed in the same session.

The new collating order remains in effect for the rest of the session, or until you issue another SET COLLATION statement. Other sessions are not affected, but database objects that you created with a non-default collation use whatever collating order was in effect at their time of their creation.

By default, the collating order is the code-set order, but some locales also support a locale-specific order. In most contexts, only NCHAR and NVARCHAR data values can be sorted according to a locale-specific collating order.

Specifying a Collating Order with SET COLLATION

SET COLLATION replaces the current collating order for NCHAR and NVARCHAR values with that of the specified *locale* for all database servers previously accessed in the current session.

For example, this statement specifies the collating order for the **GB18030-2000** code set of the Chinese language:

```
EXEC SQL set collation "zh_cn.gb18030-2000";
```

If the next operation of a database server in this session sorted NCHAR or NVARCHAR values, the result would follow the Chinese collating order that the SET COLLATION statement specified.

Suppose that in the same session, the following SET NO COLLATION statement restores the **DB_LOCALE** setting for the collating order:

```
EXEC SQL set no collation;
```

After SET NO COLLATION executes, subsequent collation in the same session is based on the **DB_LOCALE** setting. Any database objects that you created using the Chinese collating order, however, such as check constraints, indexes, prepared objects, triggers, or UDRs, will continue to apply Chinese collation rules to NCHAR and NVARCHAR data types.

Collation in an NLSCASE INSENSITIVE database

Suppose that you use the following SET COLLATION statement to replaces the current collating order with that of the **de_de.8859-1** locale for the German language:

```
SET COLLATION "de_de.8859-1";
```

By default, this locale supports case differences between uppercase and lowercase alphabetic characters. For example, two CHAR or LVARCHAR strings of the same letters and in the same order, but with differences in case, such as `'Zug'` and `'ZUG'`, are treated in LIKE or MATCHES expressions or sorted in ORDER BY clauses as distinct data values.

In a database created with the `NLSCASE INSENSITIVE` keywords, however, collating operations on NCHAR and NVARCHAR data disregard case differences, so that the database server treats case variants among strings composed of same sequence letters as duplicates. The resulting collated list orders these case-insensitive duplicates in their order of retrieval, so a collated list with case variants of the string `alpha` might appear in any order, such as this order, which disregards variations in case:

```
alpha
Alpha
alpha
ALPHA
Alpha
```

For more information, see Duplicate rows in NLSCASE INSENSITIVE databases on page 699 and NCHAR and NVARCHAR expressions in case-insensitive databases.

Restrictions on SET COLLATION

Although SET COLLATION enables you to change the collating order of the database server dynamically within a session, you should be aware of several limitations on the scope of what the SET COLLATION statement can accomplish.

- Only collation performed by the database server is affected. Client processes that sort data are not affected by SET COLLATION.
- Only the current session is affected. Other sessions are not affected directly by your SET COLLATION statements, but the database server will use their creation-time collating order to set any database objects that you create after SET COLLATION has run successfully.
- Changing the collating order does not change the code set. The database server always uses the code set specified by **DB_LOCALE**.
- Only NCHAR and NVARCHAR values are sorted in locale-specific order.

Processing characters from dissimilar code sets

Because SET COLLATION changes only the collating order, rather than the current locale or code set, you generally cannot use this statement to insert character data from different locales into the same database. You must instead use a locale that supports Unicode if the database needs to store characters from two or more languages that require inherently different code sets or code pages. For applications and for other client applications that use the HCL Informix® GLS library, databases with locales that support UTF-8 character encoding can store characters that correspond to code points from dissimilar character sets of more than one natural language, but only if all of the following conditions are satisfied:

- The **GL_USEGLU** environment variable was a set to `1` when the database server instance was started.
- The **DB_LOCALE** environment variable was set to a valid Unicode locale when the database was created.
- The **CLIENT_LOCALE** environment variable is set to valid Unicode locale that the **DB_LOCALE** setting of the database server supports.

For HCL Informix® to use the International Components for Unicode (ICU) 4.8.1 libraries to support versions of Unicode up to 6.0, the **GL_USEGLU** environment variable must be set to a value of `1` (one) in the server environment before the server is started. This setting initializes conversion routines that enable Unicode collation and SQL operations in databases that use **UTF-8** character encoding, including the Chinese **GB18030-2000** code set. This conversion applies only to databases that were created with `GL_USEGLU=1` already set.

Attention: The **GL_USEGLU** environment variable has no effect, however, on JDBC client applications, including those of the HCL Informix® JSON compatibility wire protocol listener. To support JDBC applications correctly in Unicode locales, there is no requirement that **GL_USEGLU** be set to `1` in the client or in the server environments.

Collation Performed by Database Objects

Although the database reverts to the **DB_LOCALE** collating order after the session ends (or after you execute SET NO COLLATION), objects that you create using a non-default collation persist in the database. You can create, for example, multiple indexes on the same set of columns, called *multilingual indexes*, using different collating orders that SET COLLATION specifies.

Only one clustered index, however, can exist on a given set of columns.

Only one unique constraint or primary key can exist on a given set of columns, but you can create multiple unique indexes on the same set of columns, if each index has a different collation order.

The query optimizer ignores indexes that apply any collation other than the current session collation to NCHAR or NVARCHAR columns when calculating the cost of a query.

The collating order of an attached index must be the same as that of its table, and this must be the default collating order specified by **DB_LOCALE**.

The ALTER INDEX statement cannot change the collation of an index. Any previous SET COLLATION statement is ignored when ALTER INDEX executes.

When you compare values from CHAR columns with NCHAR columns, Informix® casts the CHAR value to NCHAR, and then applies the current collation. Similarly, before comparing VARCHAR and NVARCHAR values, Informix® first casts the VARCHAR values to NVARCHAR.

When synonyms are created for remote tables or views, the participating databases must have the same collating order. Existing synonyms, however, can be used in other databases that support SET COLLATION and the collating order of the synonym, regardless of the **DB_LOCALE** setting.

Check constraints, cursors, prepared objects, triggers, and SPL routines that sort NCHAR or NVARCHAR values use the collation that was in effect at the time of their creation, if this is different from the **DB_LOCALE** setting.

The effect on performance is sensitive to how many different collations are used when creating database objects that sort in a localized order.

SET CONNECTION statement

Use the SET CONNECTION statement to reestablish a connection between an application and a database environment and to make the connection current. You can also use this statement with the DORMANT option to put the current connection in a dormant state. Use this statement with .

Syntax

(explicit id unique_232_Connect_42_seco001) unique_232_Connect_42_seco001(explicit id unique_232_Connect_42_seco002)

unique_232_Connect_42_seco002 **SET CONNECTION** { { *'connection '* | []*connection_var* | [] <Database Environment>[] **| DEFAULT** } [[] **DORMANT**] | []**CURRENT DORMANT** }

Element	Description	Restrictions	Syntax
connection	Name of the initial connection that the CONNECT statement made	The database must already exist	Quoted String on page 1202
connection_var	Host variable that contains the value of *connection*	Must be a character data type	Language specific

Usage

You can use SET CONNECTION to make a dormant connection the current connection or to make the current connection dormant.

SET CONNECTION is not valid as a prepared statement.

Making a dormant connection as the current connection

If you use the SET CONNECTION statement without the DORMANT option, *connection* must represent a dormant connection. A *dormant connection* is a connection that is established but is not current.

The SET CONNECTION statement, with no DORMANT option, makes the specified dormant connection the current one. The connection that the application specifies must be dormant. The connection that is current when the statement executes becomes dormant.

The SET CONNECTION statement in the following example makes connection `con1` the current connection and makes `con2` a dormant connection:

```
CONNECT TO 'stores_demo' AS 'con1';
...
CONNECT TO 'demo' AS 'con2';
...
SET CONNECTION 'con1';
```

A dormant connection has a *connection context* associated with it. When an application makes a dormant connection current, it reestablishes that connection to a database environment and restores its connection context. (For more information on connection context, see the CONNECT statement on page 176 statement on page CONNECT statement on page 176.) Reestablishing a connection is comparable to establishing the initial connection, except that it typically

avoids authenticating the permissions for the user again, and it avoids reallocating resources associated with the initial connection. For example, the application does not need to reprepare any statements that have previously been prepared in the connection, nor does it need to redeclare any cursors.

Making a current connection as the dormant connection

In the SET CONNECTION *connection* DORMANT statement, *connection* must represent the current connection. The SET CONNECTION statement with the DORMANT option makes the specified current connection a dormant connection.

For example, the following SET CONNECTION statement makes connection `con1` dormant:

```
SET CONNECTION 'con1' DORMANT;
```

The SET CONNECTION statement with the DORMANT option generates an error if you specify a connection that is already dormant. For example, if connection `con1` is current and connection `con2` is dormant, the following SET CONNECTION statement returns an error message:

```
SET CONNECTION 'con2' DORMANT;
```

The following SET CONNECTION statement executes successfully:

```
SET CONNECTION 'con1' DORMANT;
```

Dormant Connections in a Single-Threaded Environment

In a single-threaded application (one that does not use threads), the DORMANT option makes the current connection dormant. Using this option makes single-threaded applications upwardly compatible with thread-safe applications. A single-threaded environment, however, can have only one active connection while the program executes.

Dormant Connections in a Thread-Safe Environment

In a thread-safe application, the DORMANT option makes an active connection dormant. Another thread can now use the connection by issuing the SET CONNECTION statement without the DORMANT option. A thread-safe environment can have many threads (concurrent pieces of work performing particular tasks) in one application, and each thread can have one active connection.

An active connection is associated with a particular thread. Two threads cannot share the same active connection. Once a thread makes an active connection dormant, that connection is available to other threads. A dormant connection is still established but is not currently associated with any thread. For example, if the connection named `con1` is active in the thread named `thread_1`, the thread named `thread_2` cannot make connection `con1` its active connection until `thread_1` has made connection `con1` dormant.

The following code fragment from a thread-safe program shows how a particular thread within a thread-safe application makes a connection active, performs work on a table through this connection, and then makes the connection dormant so that other threads can use the connection:

```
thread_2()
{  /* Make con2 an active connection */
   EXEC SQL connect to 'db2' as 'con2';
   /*Do insert on table t2 in db2*/
```

```
   EXEC SQL insert into table t2 values(10);
   /* make con2 available to other threads */
   EXEC SQL set connection 'con2' dormant;
}
```

If a connection to a database environment was initiated using the CONNECT . . . WITH CONCURRENT TRANSACTION statement, any thread that subsequently connects to that database environment can use an ongoing transaction. In addition, if an open cursor is associated with such a connection, the cursor remains open when the connection is made dormant.

Threads within a thread-safe application can use the same cursor by making the associated connection current, even though only one thread can use the connection at any given time.

Identifying the Connection

If the application did not specify a connection name in the initial CONNECT statement, you must use a database environment (such as a database name or a database pathname) as the connection name. For example, the following SET CONNECTION statement uses a database environment for the connection name because the CONNECT statement does not use a connection name. For information about quoted strings that specify a database environment, see Database Environment on page 177.

```
CONNECT TO 'stores_demo';
...
CONNECT TO 'demo';
...
SET CONNECTION 'stores_demo';
```

If a connection to a database server was assigned a connection name, however, you must use the connection name to reconnect to the database server. An error is returned if you use a database environment rather than the connection name when a connection name exists.

DEFAULT Option

The DEFAULT option specifies the default connection for a SET CONNECTION statement. The default connection is one of the following connections:

- An explicit default connection (a connection established with the CONNECT TO DEFAULT statement)
- An implicit default connection (any connection established with the DATABASE or CREATE DATABASE statements)

Use SET CONNECTION without a DORMANT option to reestablish the default connection, or with that option to make the default connection dormant.

For more information, see The DEFAULT Connection Specification on page 181 and The Implicit Connection with DATABASE Statements on page 182.

CURRENT Keyword

Use the CURRENT keyword with the DORMANT option of the SET CONNECTION statement as a shorthand form of identifying the current connection. The CURRENT keyword replaces the current connection name. If the current connection is `con1`, the following two statements are equivalent:

```
SET CONNECTION 'con1' DORMANT;

SET CONNECTION CURRENT DORMANT;
```

When a Transaction is Active

Without the DORMANT keyword, the SET CONNECTION statement implicitly puts the current connection in the dormant state.

When you issue a SET CONNECTION statement with the DORMANT keyword, the SET CONNECTION statement explicitly puts the current connection in the dormant state. In both cases, the statement can fail if a connection that becomes dormant has an uncommitted transaction. If the connection that becomes dormant has an uncommitted transaction, the following conditions apply:

- If the connection was established using the WITH CONCURRENT TRANSACTION clause of the CONNECT statement, SET CONNECTION succeeds and puts the connection in a dormant state.
- If the connection was not established by the WITH CONCURRENT TRANSACTION clause of the CONNECT statement, SET CONNECTION fails and cannot set the connection to a dormant state, and the transaction in the current connection continues to be active. The statement generates an error and the application must decide whether to commit or roll back the active transaction.

SET CONSTRAINTS statement

Use the SET CONSTRAINTS statements to change how some or all of the existing constraints on a table are processed.

Syntax

Only the CREATE TABLE, CREATE TEMP TABLE and ALTER TABLE ADD CONSTRAINT statements of SQL can create new constraints. The SET CONSTRAINTS statement supports the following syntax for modifying how the database server enforces (or ignores) one or more existing constraints on a single table:

(explicit id unique_186_Connect_42_secon202) unique_186_Connect_42_secon202 **SET CONSTRAINTS** { { { *constraint* **| ALL** } { **IMMEDIATE | DEFERRED** } | [] { *constraint* | **FOR** [*'owner'* .] *table* } <Constraint Mode> } }

Constraint Mode

" { **DISABLED** | { **ENABLED** | **FILTERING** { **WITHOUT ERROR | WITH ERROR** } (explicit id) } [(explicit id) **NOVALIDATE**] } "

Element	Description	Restrictions	Syntax
constraint	Constraint whose mode is to be reset	Must exist, and must all be defined on the same table	Identifier on page 1231
owner	Owner of *table*	Must own table	Owner name on page 1262
table	Table whose constraint mode is to be reset for all constraints	Must exist in the database	Identifier on page 1231

Usage

Constraint-mode keyword options of the SET CONSTRAINTS statements include these:

- Whether constraints are checked at the statement level (`IMMEDIATE`) or at the transaction level (`DEFERRED`)
- Whether to enable (`ENABLED`) or disable (`DISABLED`) constraints
- Whether the filtering mode of constraints on tables with violations tables should be `FILTERING WITH ERROR` or `FILTERING WITHOUT ERROR`
- Whether to enable referential constraints without verifying (`NOVALIDATE`) that the foreign-key value in every row matches a primary-key value in the referenced table.
- Whether to enable check constraints without verifying (`NOVALIDATE`) that the value in every row satisfies the specified conditions.

The `SET CONSTRAINTS` keywords can begin the SET Transaction Mode statement, which is described in SET Transaction Mode statement on page 841.

The `SET CONSTRAINTS` keywords can also begin a special case of the SET Database Object Mode statement, which is an extension to the ANSI/ISO standard for SQL. Besides constraints, the SET Database Object Mode statement can also enable or disable a trigger or index, or change the filtering mode of a unique index. For the complete syntax and semantics of that statement, see SET Database Object Mode statement on page 784.

For information on using the SET CONSTRAINTS statement to enable or disable system-defined indexes that are implicitly created by PRIMARY KEY and FOREIGN KEY constraint definitions, see the topic SET INDEXES statement on page 810.

Persistence of Constraint Modes

Any changes that you make to the mode of a constraint persist until that constraint mode setting is modified again, or until that constraint or its table are dropped.

The NOVALIDATE modes, however, are exceptions, because these mode do not persist beyond the SET CONSTRAINTS statement (or beyond the ALTER TABLE ADD CONSTRAINT statement) that specified the NOVALIDATE mode.

For example, for foreign-key constraints, when the DDL statement that specifies a NOVALIDATE mode completes, the constraint mode reverts to whichever mode the **sysobjstate** system catalog table recorded for that foreign-key constraint among these three possible modes:

- `ENABLED NOVALIDATE` becomes `ENABLED`
- `FILTERING WITH ERROR NOVALIDATE` becomes `FILTERING WITH ERROR`
- `FILTERING WITHOUT ERROR NOVALIDATE` becomes `FILTERING WITHOUT ERROR`.

In all subsequent DML operations on the table, such as DELETE, INSERT, MERGE, or UPDATE statements of SQL, the database server enforces the enabled foreign-key constraint at a time determined by its `IMMEDIATE` or `DEFERRED` setting, but ignoring any previous `NOVALIDATE` mode.

Restrictions on Secondary Servers

In cluster environments, the SET CONSTRAINTS ENABLED and SET CONSTRAINTS DISABLED statements are not supported on updatable secondary servers. (More generally, session-level index, trigger, and constraint modes that the SET Database Object Mode statement specifies are not redirected for UPDATE operations on table objects in databases of secondary servers.)

SET Database Object Mode statement

Use the SET Database Object Mode statement to change the filtering mode of constraints and of unique indexes, or to enable or disable constraints, indexes, and triggers, or to make user created indexes visible/invisible, or to bypass referential-integrity checking of constraints while this statement is resetting the constraint mode.

This statement is an extension to the ANSI/ISO standard for SQL. To specify whether constraints are checked at the statement level or at the transaction level, see SET Transaction Mode statement on page 841.

Syntax

(explicit id unique_31_Connect_42_sdob001b) unique_31_Connect_42_sdob001b(explicit id unique_31_Connect_42_sdob002b)

unique_31_Connect_42_sdob002b **SET** { <Object-List Format>[] | <Table Format>[] }

Usage

In the context of this statement, *database object* has the restricted meaning of an *index*, a *trigger*, or a *constraint*, rather than the more general meaning of this term that the description of the Database Object Name on page 1225 segment defines in Other syntax segments on page 1209.

The scope of the SET Database Object Mode statement is restricted to constraints, indexes, or triggers in the local database to which the session is currently connected. After you change the mode of an object, the new mode is in effect for all sessions of that database, and persists until another SET Database Object Mode statement changes it again, or until the object is dropped from the database.

Important:

The NOVALIDATE modes to which this statement can reset foreign-key constraints are exceptions to the general statement above, as the following section of this topic explains.

Object modes for triggers, indexes and constraints

Only two object modes are available for triggers and for indexes that allow duplicate values:

- Enabled (specified by the `ENABLED` keyword)
- Disabled (specified by the `DISABLED` keyword)

For constraints and for unique indexes, you can also specify two additional modes:

- filtering without integrity-violation errors (by the `FILTERING WITHOUT ERROR` keywords)
- filtering with integrity-violation errors (by the `FILTERING WITH ERROR` keywords)

For foreign-key and check constraints, you can also specify three additional modes:

- enabled, but without checking for integrity-violation errors (by the `ENABLED NOVALIDATE` keywords)
- filtering with integrity-violation errors, but without checking for integrity-violation errors (by the `FILTERING WITH ERROR NOVALIDATE` keywords)
- filtering without integrity-violation errors, but without checking for integrity-violation errors (by the `FILTERING WITHOUT ERROR NOVALIDATE` keywords).

The constraint modes with the NOVALIDATE keyword only persist while the SET Database Object Mode statement is running, after which the constraint mode reverts to the corresponding enabled or filtering mode, and with enforcement of referential integrity or check conditions during subsequent DML operations. But for large tables that are thought to be free of referential or check constraint violations, these modes that bypass validation of the foreign-key or check constraint can significantly reduce the time required to migrate or to import large data sets.

At any given time, an object must be in exactly one of these modes. These modes, which are sometimes called *object states*, are described in the section Definitions of Database Object Modes on page 791.

The **sysobjstate** system catalog table lists all of the constraint, index, and trigger objects in the database, and the current mode of each object. Because the NOVALIDATE modes persist only during the SET CONSTRAINTS statement or ALTER TABLE ADD CONSTRAINT statement that specified that mode, the **sysobjstate** table ignores NOVALIDATE modes, which suppress violation-checking only within those DDL statements. For information on the **sysobjstate** table, see the *HCL® Informix® Guide to SQL: Reference*.

In cluster environments, the SET Database Object Mode statement is not supported on updatable secondary servers. (More generally, any session-level index, trigger, or constraint modes that the statement specifies are not redirected for UPDATE operations on table objects in databases of secondary servers.)

Privileges Required for Changing Database Object Modes

To change the mode of a constraint, index, or trigger, you must have the necessary access privileges. You must meet at least one of these requirements:

- You must have the DBA privilege on the database.
- You must be the owner of the table on which the database object is defined and you must also have the Resource privilege on the database.
- You must have the Alter privilege on the table on which the database object is defined and you must also have the Resource privilege on the database.

Object-List Format

Use the object-list format to change the mode for one or more constraint, index, or trigger.

(explicit id unique_1095_Connect_42_lmf002) unique_1095_Connect_42_lmf002(explicit id unique_1095_Connect_42_lmf003) unique_1095_Connect_42_lmf003

Object-List Format

" { **CONSTRAINTS** *constraint* <Modes for Constraints and Unique Indexes> [] | **INDEXES** *index* { <Modes for Constraints and Unique Indexes> [] | <Modes for Triggers and Duplicate Indexes> [] } | **TRIGGERS** *trigger* [<Modes for Triggers and Duplicate Indexes> []] } "

Element	Description	Restrictions	Syntax
constraint	Name of a constraint whose mode is to be set	Must be a local constraint, and all constraints in the list must be defined on the same table	Identifier on page 1231
index	Name of an index whose mode is to be set	Must be a local index, and all indexes in the list must be defined on the same table	Identifier on page 1231
trigger	Name of a trigger whose mode is to be set	Must be a local trigger, and all triggers in the list must be defined on the same table or view	Identifier on page 1231

For example, to change the mode of the unique index **unq_ssn** on the **cust_subset** table to filtering, enter the following statement:

```
SET INDEXES unq_ssn FILTERING;
```

You can also use the object-list format to change the mode for a list of constraints, indexes, or triggers that are defined on the same table. Assume that four triggers are defined on the **cust_subset** table: **insert_trig**, **update_trig**, **delete_trig**, and **execute_trig**. Also assume that all four triggers are enabled. To disable all triggers except **execute_trig**, enter this statement:

```
SET TRIGGERS insert_trig, update_trig, delete_trig DISABLED;
```

If **my_trig** is a disabled INSTEAD OF trigger on a view, the following statement enables that trigger:

```
SET TRIGGERS my_trig ENABLED;
```

In cluster environments, the SET TRIGGERS statement is not supported on updatable secondary servers. More generally, session-level index, trigger, and constraint modes that the SET Database Object Mode statement specifies are not redirected for UPDATE operations on table objects in databases of secondary servers.

Table Format

Use the table format to change the mode of all database objects of a specified type that have been defined on the same table or view.

(explicit id unique_1096_Connect_42_tamf002) unique_1096_Connect_42_tamf002(explicit id unique_1096_Connect_42_tamf003)

unique_1096_Connect_42_tamf003

Table Format

"{ | CONSTRAINTS | INDEXES | TRIGGERS }"

"FOR ['owner' .] table_object"

"{ <Modes for Constraints and Unique Indexes> | <Modes for Triggers and Duplicate Indexes> }"

Element	Description	Restrictions	Syntax
owner	Owner of *table*	Must own *table*	Owner name on page 1262
table_object	Table or view on which objects are defined	Must be a local table or view. Objects defined on a temporary table cannot be set to disabled or filtering modes.	Identifier on page 1231

This example disables all constraints defined on the **cust_subset** table:

```
SET CONSTRAINTS FOR cust_subset DISABLED;
```

In table format, you can change the modes of more than one database object type with a single statement. For example, this enables all constraints, indexes, and triggers that are defined on the **cust_subset** table:

```
SET CONSTRAINTS, INDEXES, TRIGGERS FOR cust_subset ENABLED;
```

In Informix® 10.00 and in earlier versions, you cannot use the SET TRIGGERS option of the SET Database Object Mode statement to disable an inherited trigger selectively within a table hierarchy. In this release, however, disabling a trigger on a table within a hierarchy has no effect on inherited triggers. For example, the following statement disables all triggers on the specified *subtable*, but the statement has no effect on triggers on table objects that are above or below *subtable* within a table hierarchy:

```
SET TRIGGERS FOR subtable DISABLED;
```

In cluster environments, however, the SET TRIGGERS, SET INDEXES, and SET CONSTRAINTS statements are not supported on updatable secondary servers. Session-level index, trigger, and constraint modes that the SET Database Object Mode statement specifies are not redirected for UPDATE operations on table objects in databases of secondary server

Modes for constraints and unique indexes

You can specify enabled or disabled mode for a constraint or for a unique index. For tables that the START VIOLATIONS TABLE statement has associated with a violations table and a diagnostics table, you can also use the FILTERING keyword to specify an ERROR mode for processing rows that do not comply with constraints or with unique index requirements.

When you change the mode of a foreign-key constraint to ENABLED or to FILTERING, you can optionally include the NOVALIDATE keyword. This suspends referential-integrity checking for rows that violate the constraint during execution of the SET CONSTRAINTS statement.

This is the syntax for changing the database object mode of constraints or of unique indexes in SET CONSTRAINTS or SET INDEXES statements:



Usage

If you specify no mode in the ALTER TABLE or CREATE TABLE statement that creates a constraint, the constraint is enabled by default.

Similarly, if you specify no mode in the CREATE INDEX statement that creates an index, the index is enabled by default.

There is no default mode, however, for database objects in SET Database Object Mode statements. If you specify no mode in the SET CONSTRAINTS or the SET INDEXES options of SET Database Object Mode statements, the statement fails with error -201, and the constraint mode or index mode is unchanged.

The WITHOUT ERROR and WITH ERROR filtering options support DML operations in which the database server tests whether new or modified rows violate constraints or unique indexes on the target table. How the database server processes noncompliant rows in filtering mode also depends on these factors:

- Whether a violations table and a diagnostics table are associated with the table on which the constraint or the unique index is defined.
- Whether input to the associated violations and diagnostics tables is currently enabled or disabled.

For more information, see the topics START VIOLATIONS TABLE statement on page 844 and STOP VIOLATIONS TABLE statement on page 856.

Examples of changing constraint modes and unique index modes

The following statement disables the constraint `u100_1`, so that it is still registered in the system catalog, but has no effect:

```
SET CONSTRAINTS u100_1 DISABLED;
```

If `u100_1` is an enabled unique index, rather than a constraint, then the following statement has a similar effect:

```
SET INDEXES u100_1 DISABLED;
```

The following statement enables the referential constraint `u100_1` without validating the foreign-key relationships for each row:

```
SET CONSTRAINTS u100_2 ENABLED NOVALIDATE;
```

Warning:

You can specify the new mode of a foreign key constraint as `ENABLED NOVALIDATE`, or as `FILTERING WITH ERROR NOVALIDATE` or `FILTERING WITHOUT ERROR NOVALIDATE`. This can improve performance in load operations, for example, if the data set is known to have a corresponding primary key for every row that is in scope of the foreign key constraint. It is the responsibility of the user, however, to avoid corruption of the database in subsequent DML operations. If you are not sure that the data rows are compliant,

- you should disable the foreign-key constraint,
- load the data into the new database,
- and then enable the foreign-key constrain after its table has been successfully loaded, so that the database server can validate the referential integrity of the data.

The database server automatically drops the NOVALIDATE attribute when the SET CONSTRAINTS statement completes execution. The following statement enables the same foreign-key constraint and restores automatic validation of the constraint:

```
SET CONSTRAINTS u100_2 ENABLE;
```

When you use the `FILTERING WITHOUT ERROR` keywords to define a filtering mode, subsequent violations of that constraint, or uniqueness violations of that index, do not cause INSERT, DELETE, MERGE, or UPDATE operations to fail if some rows violate the constraint or the unique index. In this filtering mode, the DML statement succeeds, but the database server enforces the constraint or the unique index requirement by writing the noncompliant rows to the violations table.

The following statement instructs the database server to write any rows that violate the `r104_11` constraint to the violations table, provided that a violations table is associated with the target table.

```
SET CONSTRAINTS r104_11 FILTERING WITHOUT ERROR;
```

For more information about filtering modes, see the topic Filtering Modes on page 793.

The following statement disables all constraints defined on the `orders` table:

```
SET CONSTRAINTS FOR orders DISABLED;
```

Subsequent DML operations on that table ignore rows that violate constraints on the `orders` table, creating no entries in its violations or diagnostics tables, if those tables exist. If any unique indexes exist on the `orders` table, however, rows that violate uniqueness requirements are processed according to the current modes of the indexes, as listed in the **`sysobjstate`** system catalog table.

Enabling foreign-key constraints when an index exists on the referenced table

By default, the database server automatically validates referential constraints when their mode is changed to ENABLED. You might be able to save time when the SET CONSTRAINTS statement enables a foreign-key constraint, if the referenced table already has a unique index or a primary-key constraint on the column (or on the set of columns) corresponding to the key of the foreign-key constraint.

The database server makes a cost-based decision on how to validate the enabled foreign-key constraint. The index-key algorithm might be faster in many contexts, because it validates the constraint by scanning only the index values, rather than by scanning all the rows in the table.

The database server can consider using the index-key algorithm to validate the foreign-key constraint that it enables, but only if all of the following conditions are satisfied when the SET CONSTRAINTS ENABLED statement resets the constraint mode:

- The SET CONSTRAINTS statement is enabling only one foreign-key constraint.

  If this is the case, the database server needs to check individual values for only the column on which the foreign-key constraint is being enabled. Validating two foreign-key constraints at the same time would require two indices to be used on the same scan, which is not supported.
- The same statement is not enabling a CHECK constraint.

  If the SET CONSTRAINTS statement is enabling more than one constraint, validating CHECK constraints requires that every row be checked, rather than individual values. In that case, the index-key algorithm cannot be used for validating the foreign-key constraint.
- The foreign-key columns do not include user-defined data types (UDTs) or built-in opaque data types.

  To make the fast index-key algorithm as efficient as possible, it eliminates all the inefficiencies of executing routines associated with user-defined or built-in opaque data types, such as the BOOLEAN and LVARCHAR built-in opaque types.
- The new mode of the foreign-key constraint is not DISABLED.

  If it is disabled, then no constraint-checking algorithm is needed, because no checking for referential integrity violations occurs.
- The table is not associated with an active violation table.

  Violations tables require that at the time of checking, every row that does not satisfy the new constraint must be inserted into the violation table. Scanning every row for violations prevents the database server from using the faster index-key algorithm that skips duplicate rows.

Except in the case of one or more violating rows, the SET CONSTRAINTS statement can enable and validate a foreign-key constraint when some of these requirements are not satisfied, but the database server will not consider using the index-key algorithm to validate the foreign-key constraint. The additional validation costs for scanning the entire table are generally proportional to the size of the table. These costs can be substantial for very large tables.

When you enable a self-referencing foreign-key constraint, whose REFERENCING clause specifies the same table on which the constraint is defined, the database server can consider an index-key algorithm for validating referential-integrity, if all the conditions listed above are satisfied.

Modes for Triggers and Duplicate Indexes

You can specify ENABLED or DISABLED as the only object modes for triggers on tables, and for triggers on views, and for indexes that allow duplicate values.

You can specify the modes for triggers or duplicate indexes.

Modes for Triggers and Duplicate Indexes

"{ ENABLED | DISABLED }"

If you specify no mode for an index or for a trigger when you create it or in a subsequent SET Database Object Mode statement, the object is enabled by default.

Example of changing the mode of an index

The following example of the SET INDEXES statement disables the unique **acc_num_ix** index. which was created in ENABLED mode by default:

```
CREATE TABLE accounts (
 acc_num INTEGER DEFAULT 1,
 acc_type CHAR(1) DEFAULT 'A',
 acc_descr CHAR(20) DEFAULT 'New Account',
 acc_bal MONEY(9,2)
 acc_id CHAR(32) DEFAULT CURRENT_USER);

CREATE DISTINCT INDEX acc_num_ix ON accounts (accr_num);

SET INDEXES acc_num_ix DISABLED;
```

Here the SET INDEXES statement used the Object-list format to specify the index by name. Because no other index is defined on the **accounts** table, the following SET INDEXES statement that uses the Table format has the same effect:

```
SET INDEXES FOR accounts DISABLED;
```

Example of changing mode of a trigger

The following example enables a **backup_accts** trigger that was created in DISABLED mode:

```
SET TRIGGERS backup_accts ENABLED;
```

Definitions of Database Object Modes

You can use database object modes to control the effects of INSERT, DELETE, and UPDATE statements. Your choice of mode affects the tables whose data you are manipulating, the behavior of the database objects defined on those tables, and the behavior of the data manipulation statements themselves.

Enabled Mode

Database objects in the enabled mode behave as constraints, indexes, or triggers during DML operations on the table.

If you specify no database object mode when constraints, indexes, or triggers are created, they are enabled by default. The data definition statements CREATE TABLE, ALTER TABLE, CREATE INDEX, and CREATE TRIGGER all create database objects in enabled mode, unless you explicitly specify a different mode.

Which nondefault object modes are available at creation-time depends on the type of object:

- When a trigger or a non-unique index is created, the only keyword alternative to the enabled mode is `DISABLED`.
- When a constraint or a unique index is created, alternatives to the default or explicit `ENABLED` keyword include `DISABLED`, `FILTERING WITH ERROR`, and `FILTERING WITHOUT ERROR`. (But if you only specify `FILTERING`, then `FILTERING WITHOUT ERROR` is the default error mode for `FILTERING` objects.)
- While the ALTER TABLE ADD CONSTRAINT statement is creating a foreign-key or check constraint, however, any of these three modes can instead be specified as additional alternatives to the enabled mode:
  - `ENABLED NOVALIDATE`
  - `FILTERING WITH ERROR NOVALIDATE`
  - `FILTERING WITHOUT ERROR NOVALIDATE`.

When the SET Database Object Mode statement changes the mode of an existing constraint, index, or trigger, however, there no default mode. If you specify no object mode, the SET Database Object Mode statement fails with error -201. If you want to reset the mode of a constraint, index, or trigger to enabled from some other mode, you must explicitly specify the `ENABLED` keyword.

When a database object is successfully enabled, the database server registers that object state in the **sysobjstate** table of the system catalog, and takes that database object into consideration when its table is the target of a subsequent INSERT, DELETE, MERGE, or UPDATE statement (or for Select triggers, a SELECT statement). Thus, an enabled constraint is enforced, an enabled index is updated, and an enabled trigger on a table is executed when the trigger event takes place.

For example, after you set foreign-key constraints and unique indexes to enabled mode, when an INSERT, DELETE, MERGE, or UPDATE operation attempts to violate the referential integrity of the table, the data manipulation operation fails, no rows in the table are changed, and the database server returns an error message.

ENABLED NOVALIDATE mode for foreign-key or check constraints

While the SET Database Object Mode statement is changing the mode of a foreign-key or check constraint to ENABLED, the database server validates the constraint by examining every row in the constrained table to verify the constraint conditions. This validation can require significant time and resources. You can instead bypass the search for violating rows during the SET Database Object mode operation by including the `NOVALIDATE` keyword to change the constraint mode to ENABLED NOVALIDATE. For large tables, specifying `ENABLED NOVALIDATE` can substantially reduce the time required to enable the constraint.

After the SET CONSTRAINTS option to the SET Database Object Mode statement successfully enables a foreign-key or check constraint, the constraint mode is registered as enabled (`E`) in the **sysobjstate** system catalog table. The `NOVALIDATE`

keyword, that had prevented checking for referential-integrity or check condition violations while the SET CONSTRAINTS statement was running, is not encoded anywhere in the system catalog, and has no further effect on the object mode or the behavior of the constraint.

Until it is dropped or disabled, that constraint is enforced during subsequent DML operations on its table, in order to maintain the referential integrity of the database.

Disabled Mode

When a database object is disabled, the database server ignores it during the execution of an INSERT, DELETE, MERGE, SELECT, or UPDATE statement. A disabled constraint is not enforced, a disabled index is not updated, and a disabled trigger is not executed when the trigger event takes place.

When you disable constraints and unique indexes, any data manipulation statement that violates the restriction of the constraint or unique index succeeds (that is, the target row is changed), and the database server does not return an error message.

You can use the disabled mode to add a new constraint or new unique index to an existing table, even if some rows in the table do not satisfy the new integrity specification. Disabling can also be efficient in LOAD operations.

For information on adding a constraint, see Adding a Constraint That Existing Rows Violate on page 148 in the ALTER TABLE statement. For information on adding a unique index, see Adding a Unique Index When Duplicate Values Exist in the Column on page 259 in the CREATE INDEX statement.

Filtering Modes

A constraint or unique index in a filtering mode can insert into an associated violations table any rows that fail to comply with the constraint or index during DML operation. This mode also supports WITH ERROR and WITHOUT ERROR options for processing referential-integrity violations from INSERT, DELETE, MERGE, and UPDATE statements.

When a constraint or unique index is in FILTERING WITH ERROR mode, the database server returns a referential-integrity violation error message after the INSERT, DELETE, MERGE, or UPDATE statement results in one or more rows that are not in compliance with the unique index or with the constraint.

By default, the `FILTERING` keyword with no error option specifies the FILTERING WITHOUT ERROR object mode.

Effects of FILTERING mode in DML operations

When a constraint or unique index is in FILTERING WITHOUT ERROR mode, the INSERT, DELETE, MERGE, or UPDATE statement succeeds, but the database server enforces the constraint or the unique-index requirement by writing any failed rows to the violations table associated with the target table. Diagnostic information about the constraint violation or unique-index violation is written to the diagnostics table associated with the target table.

In data manipulation operations, filtering mode has the following specific effects on INSERT, UPDATE, and DELETE statements:

- A constraint violation during an INSERT statement causes the database server to make a copy of the nonconforming record and write it to the violations table. The database server does not write the nonconforming record to the target table.

  If the INSERT statement is not a singleton INSERT, the rest of the insert operation proceeds with the next record.
- A constraint violation or unique-index violation during an UPDATE statement causes the database server to make a copy of the existing record that was to be updated and write it to the violations table. The database server also makes a copy of the new record and writes it to the violations table, but the actual record is not updated in the target table. If the UPDATE statement is not a singleton update, the rest of the update operation proceeds with the next record.
- A constraint violation or unique-index violation during a DELETE statement causes the database server to make a copy of the record that was to be deleted and write it to the violations table. The database server does not delete the actual record in the target table. If the DELETE statement is not a singleton delete, the rest of the delete operation proceeds with the next record.
- In MERGE statements, the component INSERT, DELETE, or UPDATE operations are processed as respectively described above.

In all of these cases, the database server sends diagnostic information about each constraint violation or unique-index violation to the diagnostics table associated with the target table.

For information on the structure of the records that the database server writes to the violations and diagnostics tables, see Structure of the violations table on page 847 and Structure of the diagnostics table on page 853.

FILTERING NOVALIDATE modes

While the SET Database Object Mode statement is changing the mode of a foreign-key or check constraint to FILTERING WITHOUT ERROR or to FILTERING WITH ERROR, the database server validates the constraint by examining every row in the constrained table. For large tables, this validation can require significant time and resources.

You can instead bypass the search for violating rows during the SET Database Object mode operation by including the `NOVALIDATE` keyword to change the constraint mode to FILTERING WITHOUT ERROR NOVALIDATE or to FILTERING WITH ERROR NOVALIDATE, as in these examples for foreign-key constraints:

```
SET CONSTRAINTS refcon_1, refcon_2 FILTERING WITH ERROR;
SET CONSTRAINTS refcon_3, refcon_4 FILTERING WITHOUT ERROR;
```

For constraints on large tables that need to be relocated, specifying `NOVALIDATE` for ENABLED or FILTERING modes can substantially reduce the time required to change the mode of the constraint to a filtering mode.

After the SET Database Object Mode statement successfully enables a constraint in a NOVALIDATE filtering mode, the constraint is registered in FILTERING WITHOUT ERROR mode (**F**) or in FILTERING WITH ERROR mode (**G**) in the **sysobjstate** system catalog table. Because the `NOVALIDATE` keyword has no encoding in the system catalog, it has no subsequent effect on the behavior of the database server. The database server enforces the foreign-key constraint during subsequent DML operations as the SET CONSTRAINTS statement specified, with or without integrity violation errors, to maintain the referential integrity of the database.

Important:

When the ALTER TABLE ADD CONSTRAINT statement defines an ENABLED referential constraint in NOVALIDATE mode on a table that contains data, the database server can achieve the same efficiencies of bypassing constraint validation that are described above for the SET CONSTRAINTS statement. For more information about the `NOVALIDATE` option in ALTER TABLE operations that define new referential constraints, see Creating foreign-key constraints in NOVALIDATE modes on page 123.

Starting and Stopping the Violations and Diagnostics Tables

About this task

You must use the START VIOLATIONS TABLE statement to start the violations and diagnostics tables for the target table on which the database objects are defined, either before you set any database objects that are defined on the table to the filtering mode, or after you set database objects to filtering, but before any users issue INSERT, DELETE, or UPDATE statements.

If you want to stop the database server from filtering bad records to the violations table and sending diagnostic information about each bad record to the diagnostics table, you must issue a STOP VIOLATIONS TABLE statement.

For further information on these statements, see START VIOLATIONS TABLE statement on page 844 and STOP VIOLATIONS TABLE statement on page 856.

Error Options for Filtering Mode

When you set the mode of a constraint or unique index to filtering, you can specify one of two error options. These error options control whether the database server displays an integrity-violation error message when it encounters bad records during execution of data manipulation statements:

- The WITH ERROR option instructs the database server to return a referential integrity-violation error message after executing an INSERT, DELETE, or UPDATE statement in which one or more of the target rows causes a constraint violation or a unique-index violation.
- The WITHOUT ERROR option is the default. This option prevents the database server from issuing a referential integrity-violation error message to the user after an INSERT, DELETE, or UPDATE statement causes a constraint violation or a unique-index violation.

Effect of Filtering Mode on the Database

The net effect of the filtering mode is that the contents of the target table always satisfy all constraints on the table and any unique-index requirements on the table.

In addition, the database server does not lose any data values that violate a constraint or unique-index requirement, because non-conforming records are sent to the violations table, and diagnostic information about those records is sent to the diagnostics table.

Furthermore, when filtering mode is in effect, insert, delete, and update operations on the target table do not fail when the database server encounters bad records. These operations succeed in adding all the good records to the target table. Thus, filtering mode is appropriate for large-scale batch updates of tables. The user can fix records that violate constraints and unique-index requirements after the fact. The user does not need to fix the bad records before the batch update to avoid losing the bad records during the batch update.

SET DATASKIP statement

Use the SET DATASKIP statement to control whether the database server skips a dbspace that is unavailable during the processing of a transaction.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
SET DATASKIP { ON [ dbspace ] | OFF | DEFAULT }
```

Element	Description	Restrictions	Syntax
dbspace	Name of the skipped dbspace	Must exist at time of execution	Identifier on page 1231

Usage

SET DATASKIP allows you to reset at runtime the Dataskip feature, which controls whether the database server skips a dbspace that is unavailable (for example, due to a media failure) in the course of processing a transaction.

In , the warning flag **sqlca.sqlwarn.sqlwarn6** is set to `w` if a dbspace is skipped. See also the *HCL® Informix® Enterprise Replication Guide*.

In Informix®, this statement applies only to tables that are fragmented across dbspaces or partitions. It does not apply to blobspaces nor to sbspaces.

Specifying SET DATASKIP ON without including a *dbspace* instructs the database server to skip any dbspaces in the fragmentation list that are unavailable. You can use the **onstat** **-d** or **-D** options to determine whether a dbspace is down.

When you specify SET DATASKIP ON *dbspace*, you are instructing the database server to skip the specified *dbspace* if it is unavailable.

If you specify SET DATASKIP OFF, the Dataskip feature is disabled. If you specify SET DATASKIP DEFAULT, the database server uses the setting that is specified in the DATASKIP configuration parameter in ONCONFIG file.

Examples

The following skips **dbsp1** for the current session:

```
SET DATASKIP ON dbsp1;
```

The following sets the value of **DATASKIP** to the value specified in **onconfig**:

```
SET DATASKIP DEFAULT;
```

The following switches **DATASKIP** off so that all dbspaces are used.

```
SET DATASKIP OFF;
```

Circumstances When a Dbspace Cannot Be Skipped

The database server cannot skip a dbspace under certain conditions. The following list outlines those conditions:

- Referential constraint checking

  When you want to delete a parent row, the child rows must also be available for deletion, and must exist in an available fragment.

  When you want to insert a new child row, the parent row must be found in the available fragments.

- Updates

  When you perform an update that moves a record from one fragment to another, both fragments must be available.

- Inserts

  When you try to insert records in a expression-based fragmentation strategy and the dbspace is unavailable, an error is returned.

  When you try to insert records in a round-robin fragment-based strategy, and a dbspace is down, the database server inserts the rows into any available dbspace.

  When no dbspace is available, an error is returned.

- Indexing

  When you perform updates that affect the index, such as when you insert or delete rows, or update an indexed column, the index must be available.

  When you try to create an index, the dbspace you want to use must be available.

- Serial keys

  The first fragment is used to store the current serial-key value internally. This is not visible to you except when the first fragment becomes unavailable and a new serial key value is required, which can happen during INSERT statements.

SET DEBUG FILE statement

Use the SET DEBUG FILE statement to identify the file that is to receive the runtime trace output of an SPL routine.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

SET DEBUG FILE TO { *'filename'* | *filename_var* | *expression* } [**WITH APPEND**]

Element	Description	Restrictions	Syntax
expression	Expression that returns a filename	Must be a valid filename	Expression on page 1003
filename	Pathname of the file that contains the output of the TRACE statement	See Using the WITH APPEND Option on page 798	Quoted String on page 1202.
filename_var	Host variable storing *filename* string	Must be a character type	Language specific

Usage

This statement specifies the file to which the database server writes the output from the TRACE statement in the SPL routine. Each time the TRACE statement is executed, the trace information is added to this output file.

Using the WITH APPEND Option

The output file that you specify in the SET DEBUG FILE statement can be a new file or existing file. If you specify an existing file, its current contents are deleted when you issue the SET DEBUG FILE TO statement. The first execution of a TRACE statement sends trace output to the beginning of the file.

If you include the WITH APPEND option, the current contents of the file are preserved when you issue the SET DEBUG FILE statement. The first execution of a TRACE statement adds the new trace output to the end of the file.

If you specify a new file in the SET DEBUG FILE TO statement, it makes no difference whether you include the WITH APPEND option. The first execution of a TRACE statement sends trace output to the beginning of the new file whether you include or omit the WITH APPEND option.

Closing the Output File

About this task

To close the file that the SET DEBUG FILE TO statement opened, issue another SET DEBUG FILE TO statement with another filename. You can then read or edit the contents of the first file.

Redirecting Trace Output

You can use the SET DEBUG FILE TO statement outside an SPL routine to direct the trace output of the SPL routine to a file. You can also use this statement within an SPL routine to redirect its own output.

Location of the Output File

If you execute the SET DEBUG FILE statement with a simple filename on a local database, the output file is located in your current directory. If your current database is on a remote database server, the output file is located in your home directory on the remote database server. If you provide a full pathname for the debug file, the file is placed in the directory that you specify on the remote database server. If you do not have write permissions in the directory, you receive an error.

The following example sends the output of the SET DEBUG FILE TO statement to a file called **debug.out**:

```
SET DEBUG FILE TO 'debug' || '.out';
```

SET DEFERRED_PREPARE statement

Use the SET DEFERRED_PREPARE statement to control whether a client process postpones sending a PREPARE statement to the database server until the OPEN or EXECUTE statement is sent.

Only supports this statement, which is an extension to the ANSI/ISO standard for SQL. You can use this statement only with .

Syntax

```
SET DEFERRED_PREPARE { ENABLED | [ { DISABLED } ] }
```

Usage

By default, the SET DEFERRED_PREPARE statement causes the application program to delay sending the PREPARE statement to the database server until the OPEN or EXECUTE statement is executed. In effect, the PREPARE statement is bundled with the other statement so that one round-trip of messages, instead of two, is sent between the client and the server. This Deferred-Prepare feature can affect the following series of Dynamic SQL statement:

- PREPARE, DECLARE, OPEN statement blocks that operate with the FETCH or PUT statements
- PREPARE followed by the EXECUTE or EXECUTE IMMEDIATE statement

You can specify ENABLED or DISABLED options for SET DEFERRED_PREPARE.

If you specify no option, the default is ENABLED. The following example enables the Deferred-Prepare feature by default:

```
EXEC SQL set deferred_prepare;
```

The ENABLED option enables the Deferred-Prepare feature within the application. The following example explicitly specifies the ENABLED option:

```
EXEC SQL set deferred_prepare enabled;
```

After an application issues SET DEFERRED_PREPARE ENABLED, the Deferred-Prepare feature is enabled for subsequent PREPARE statements in the application. The application then exhibits the following behavior:

- The sequence PREPARE, DECLARE, OPEN sends the PREPARE statement to the database server with the OPEN statement. If the prepared statement has syntax errors, the database server does not return error messages to the application until the application declares a cursor for the prepared statement and opens the cursor.
- The sequence PREPARE, EXECUTE sends the PREPARE statement to the database server with the EXECUTE statement. If a prepared statement contains syntax errors, the database server does not return error messages to the application until the application attempts to execute the prepared statement.

If Deferred-Prepare is enabled in a PREPARE, DECLARE, OPEN statement block that contains a DESCRIBE statement, the DESCRIBE statement must follow the OPEN statement rather than the PREPARE statement. If the DESCRIBE follows PREPARE, the DESCRIBE statement results in an error.

Use the DISABLED option to disable the Deferred-Prepare feature within the application. The following example specifies the DISABLED option:

```
EXEC SQL set deferred_prepare disabled;
```

If you specify the DISABLED option, the application sends each PREPARE statement to the database server when the PREPARE statement is executed.

Example of SET DEFERRED_PREPARE

The following code fragment shows a SET DEFERRED_PREPARE statement with a PREPARE, EXECUTE statement block. In this case, the database server executes the PREPARE and EXECUTE statements all at once:

```
EXEC SQL BEGIN DECLARE SECTION;
   int a;
EXEC SQL END DECLARE SECTION;
EXEC SQL allocate descriptor 'desc';
EXEC SQL create database test;
EXEC SQL create table x (a int);

/* Enable Deferred-Prepare feature */
EXEC SQL set deferred_prepare enabled;
/* Prepare an INSERT statement */
EXEC SQL prepare ins_stmt from 'insert into x values(?)';
a = 2;
EXEC SQL EXECUTE ins_stmt using :a;
if (SQLCODE)
   printf("EXECUTE : SQLCODE is %d\n", SQLCODE);
```

Using Deferred-Prepare with OPTOFC

You can use the Deferred-Prepare and Open-Fetch-Close Optimization (OPTOFC) features in combination. The OPTOFC feature delays sending the OPEN message to the database server until the FETCH message is sent. The following situations occur if you enable the Deferred-Prepare and OPTOFC features at the same time:

- If the text of a prepared statement contains syntax errors, the error messages are not returned to the application until the first FETCH statement is executed.
- A DESCRIBE statement cannot be executed until after the FETCH statement.
- You must issue an ALLOCATE DESCRIPTOR statement before a DESCRIBE or GET DESCRIPTOR statement can be executed.

The database server performs an internal execution of a SET DESCRIPTOR statement which sets the TYPE, LENGTH, DATA, and other fields in the system descriptor area. You can specify a GET DESCRIPTOR statement after the FETCH statement to see the data that is returned.

SET EXPLAIN statement

Use the SET EXPLAIN statement to enable or disable the recording measurements of queries in the current session, including the plan of the query optimizer, an estimate of the number of rows returned, and the relative cost of the query.

Syntax

```
SET EXPLAIN { { OFF | ON [ AVOID_EXECUTE ] } | FILE TO { expression | 'file_name' | file_name_var } }
```

Element	Description	Restrictions	Syntax
expression	Expression that returns a file name specification	Must return a string satisfying the restrictions on the file name	Expression on page 1003
file_name	The explain output file name. If the file's absolute path is not included, the explain output file will be created in the default explain output file location	Must conform to operating-system rules. If the file already exists, explain output will be appended to it.	Quoted String on page 1202
file_name_var	Host variable that stores a file name	Must be a character data type	Language specific

Usage

Output from a SET EXPLAIN ON statement is directed to the appropriate file until you issue a SET EXPLAIN OFF statement or until the program ends. If you do not enter a SET EXPLAIN statement, then the default behavior is OFF, and the database server does not generate measurements for queries.

The SET EXPLAIN statement executes during the database server optimization phase, which occurs when you initiate a query. For queries that are associated with a cursor, if the query is prepared and does not have host variables, optimization occurs when you prepare it. Otherwise, optimization occurs when you open the cursor.

The SET EXPLAIN statement provides various measurements of the work involved in performing a query.

Option

Effect

ON

Generates measurements for each subsequent query and writes the results to an output file in the current directory. If the file already exists, new output is appended to the existing file.

AVOID_EXECUTE

Prevents a SELECT, INSERT, MERGE, UPDATE, or DELETE statement from executing. The database server prints the query plan to an output file

OFF

Terminates activity of the SET EXPLAIN statement, so that measurements for subsequent queries are no longer generated or written to the output file

FILE TO

Generates measurements for each subsequent query and allows you to specify the location for the explain output file.

The following example writes the query plan in the explain output file for subsequent queries in the current session:

```
SET EXPLAIN ON;
```

The following example suspends writing query plans to a file in the current session:

```
SET EXPLAIN OFF;
```

Using the AVOID_EXECUTE Option

The AVOID_EXECUTE keyword prevents DML statements from executing. Instead, the database server prints the query plan to an output file.

The SET EXPLAIN ON AVOID_EXECUTE statement activates the Avoid Execute option for a session, or until the next SET EXPLAIN OFF (or ON) without AVOID_EXECUTE. If you activate AVOID_EXECUTE for a query that contains a remote table, the query does not execute at either the local or remote site.

The following example stores the output in the specified file.

```
SET EXPLAIN ON AVOID_EXECUTE;
SET EXPLAIN FILE TO '/tmp/explain.out';
```

When AVOID_EXECUTE is set, the database server sends a warning message. If you are using DB-Access, it displays a text message

```
Warning! avoid_execute has been set
```

for any select, delete, update or insert query operations. From ESQL, the **sqlwarn.sqlwarn7** character is set to 'W'.

Use the SET EXPLAIN ON or the SET EXPLAIN OFF statement to turn off the AVOID_EXECUTE option. The SET EXPLAIN ON statement turns off the AVOID_EXECUTE option but continues to generate a query plan and writes the results to an output file.

If you issue the SET EXPLAIN ON AVOID_EXECUTE statement in an SPL routine, the SPL routine and any DDL statements still execute, but the DML statements inside the SPL routine do not execute. The database server prints the query plan of the SPL routine to an output file. To turn off this option, you must execute the SET EXPLAIN ON or the SET EXPLAIN OFF statement outside the SPL routine. If you execute the SET EXPLAIN ON AVOID_EXECUTE statement before you execute an SPL routine, the DML statements inside the SPL routine do not execute, and the database server does not print a query plan of the SPL routine to an output file.

Nonvariant functions in a query are still evaluated when AVOID_EXECUTE is in effect, because the database server calculates these functions before optimization.

For example, the **func( )** function is evaluated, even though the following SELECT statement is not executed:

```
SELECT * FROM orders WHERE func(10) > 5;
```

For other performance implications of the AVOID_EXECUTE option, see your .

If you execute the SET EXPLAIN ON AVOID_EXECUTE statement before you open a cursor in the program, each FETCH operation returns the message that the row was not found. If you execute SET EXPLAIN ON AVOID_EXECUTE after the program opens a cursor, however, this statement has no effect on the cursor, which continues to return rows.

Using the FILE TO option

When you execute a SET EXPLAIN FILE TO statement, explain output is turned on. The SET EXPLAIN FILE TO statement can change the default file name for the explain output until the end of the session or until another SET EXPLAIN statement is issued.

The filename can be any valid combination of path and file name. If no path component is specified, the file is placed in the default explain output location. The permissions for the file are owned by the current user.

The output file that you specify in the SET EXPLAIN statement can be a new file or an existing file. If the FILE TO clause specifies an existing file, the new output is appended to that file.

Default name and location of the explain output file on UNIX™

When you issue the SET EXPLAIN ON statement, the plan that the optimizer chooses for each subsequent query is written to the explain output file.

If the explain output file does not exist when you issue SET EXPLAIN ON, the database server creates the file. If the explain output file already exists when you issue the SET EXPLAIN ON statement, subsequent output is appended to the file.

Default name of the explain output file

Explain output files generated by a SET EXPLAIN statement and explain files generated by onmode -Y have different default names. Explain output filenames for mapped users are different than the explain output filenames for OS users, as well. The following table shows the default names:

Table 13. Default explain output file names.

User and generation type	File name
Regular user and SET EXPLAIN	sqexplain.out
Mapped user and SET EXPLAIN	*username*_sqexplain.out
Regular user and onmode -Y	sqexplain.out.*session_id*
Mapped user and onmode -Y	*username*_sqexplain.out.*session_id*

Default location of the explain output file

If the client application and the database server are on the same computer, the output file is stored in your current directory. If you are using a version 5.x or earlier client application and the output file does not appear in the current directory, check your home directory for the file. When the current database is on another computer, the output file is stored in your home directory on the remote host.

For a mapped user without a home directory, the explain output file is stored in `$INFORMIXDIR/users/server_svrnum/uid_uid`.

For a mapped user with a home directory, remote clients' explain output files are stored in the user's home directory, and local clients' explain output files are stored in the user's current working directory.

Default name and location of the output file on Windows™

On Windows™, SET EXPLAIN ON writes the plan that the optimizer chooses for each subsequent query to a file in %INFORMIXDIR%\sqexpln.

For explain output files generated by the SET EXPLAIN statement, the default file name is *user_name*`.out`, where *user_name* is the user login.

For explain output files generated by onmode -Y, the default file name is `sqexplain.out.`*session_id*.

SET EXPLAIN output

View the SET EXPLAIN output file to analyze information on an executed query, including the directives set for the query, an estimate of the cost of the query, an estimate of the number of returned rows, the order in which the server accessed tables, index keys, join methods, and query statistics.

The following table lists terms that can appear in the output file and their significance.

Table 14. Output file terms

Term	Significance
Query	Displays the executed query and indicates whether SET OPTIMIZATION was set to HIGH or LOW. If you SET OPTIMIZATION to LOW, the output displays the following uppercase string as the first line: `QUERY:{LOW}` If you SET OPTIMIZATION to HIGH, the output of SET EXPLAIN displays the following uppercase string as the first line: `QUERY:`
Directives followed	Lists the directives set for the query If the syntax for a directive is incorrect, the query is processed without the directive. In that case, the output shows `DIRECTIVES NOT FOLLOWED` in addition to `DIRECTIVES FOLLOWED`. For more information on the directives specified after this term, see the Optimizer Directives on page 1246 or SET OPTIMIZATION statement on page 820. If a DELETE or UPDATE statement specifies an uncorrelated subquery in the WHERE clause, the set of qualifying rows returned by the subquery is materialized as a temporary table, and the output of SET EXPLAIN displays within parentheses the following message: `(Temp Table For Subquery)`
Estimated cost	An estimate of the amount of work for the query The optimizer uses an estimate to compare the cost of one path with another. The estimate is a number the optimizer assigns to the selected access method. This number does not translate

Table 14. Output file terms (continued)

Term	Significance
	directly into time and cannot be used to compare different queries. It can be used, however, to compare changes made for the same query. When data distributions are used, a query with a higher estimate generally takes longer to run than one with a smaller estimate. In the case of a query and a subquery, two estimated cost figures are returned; the query figure also includes the subquery cost. The subquery cost is shown so that you can see the cost that is associated with only the subquery.
Estimated number of rows returned	An estimate of the number of rows to be returned This number is based on information in the system catalog tables.
Numbered list	The order in which tables are accessed, followed by the access method used (index path or sequential scan) When a query involves table inheritance, all the tables are listed under the supertable in the order in which they were accessed.

Index name	The name of the index
	
	For example, `idx1` is the name of the following index:
	
	`Index Name: informix.idx1`
	
	`FOT` in the index name identifies the index as a forest of trees index: For example, the following index is a forest of trees index:
	
	`Index Name: informix.fot_idx (FOT)`

Index keys	The columns used as filters or indexes; the column name used for the index path or filter is indicated. The notation *(Key Only)* indicates that all the desired columns are part of the index key, so a key-only read of the index could be substituted for a read of the actual table. In databases that have the NLSCASE INSENSITIVE property, all index scan methods (except key-only scans) allow the query execution plan to map all case-sensitive values to a single value for NCHAR and NVARCHAR columns. For more on NLSCASE INSENSITIVE databases, see Duplicate rows in NLSCASE INSENSITIVE databases on page 699. The *Lower Index Filter* shows the key value where the index read begins; and the *Upper Index Filter* is shown for the key value where the index read stops. The *Index Key Filters* show filters that will be applied on retrieved index key values. If the query uses an index self-join path, the *Index Self Join Keys* shows the leading index key columns used as self-join keys, and the *Lower bound* and *Upper bound* show the boundaries of the leading index key columns.

Table 14. Output file terms (continued)

Term	Significance
Join method	When the query involves a join between two tables, the join method that the optimizer used (Nested Loop or Dynamic Hash) is shown at the bottom of the output for that query. When the query involves a dynamic join of two tables, if the output contains the words *Build Outer*, the hash table is built on the first table listed (called the build table). If the words *Build Outer* do not appear, the hash table is built on the second table listed.
Query statistics	When the EXPLAIN_STAT configuration parameter is set to 1, this section shows the number of rows returned, the number of rows estimated in the query plan, the time required, calls to iterator functions, and the estimated cost of scan and join operations on table objects.
Time	When the output displays the elapsed time for a query execution plan or for a component of that plan, the value is formatted as `minutes:seconds.fraction` to display the minutes, seconds, and fractional part of a second.

If the query uses a collating order other than the default for the **DB_LOCALE** setting, then the **DB_LOCALE** setting and the name of the other locale that is the basis for the collation in the query (as specified by the SET COLLATION statement) are both included in the output file. Similarly, if an index is not used because of its collation, the output file indicates this.

Complete-Connection Level Settings and Output Examples

The SET EXPLAIN statement supports *complete-connection level* settings.

The SET EXPLAIN statement supports *complete-connection level* settings. This means that values in the local session environment at the time of connection are propagated to all new or resumed transactions of the following types:

- transactions within the local database
- distributed transactions across databases of the same server instance
- distributed transactions across databases of two or more database server instances
- global transactions with XA-compliant data sources that are registered in the local database

If you change the SET EXPLAIN setting within a transaction, the new value is propagated back to the local environment and also to all subsequent new or resumed transactions.

Examples of SET EXPLAIN Output

The following SQL statements cause the database server to write the query plans of the UPDATE statement (and of its subquery) to the default explain output file:

```
DATABASE stores_demo;
SET EXPLAIN ON;
UPDATE orders SET ship_charge = ship_charge + 2.00
   WHERE customer_num IN
      (SELECT orders.customer_num FROM orders
```

```
          WHERE orders.ship_weight < 50);
CLOSE DATABASE;
```

The following information is displayed in the resulting output:

```
QUERY:
------
update orders set ship_charge = ship_charge + 2.00
where customer_num in
(select orders.customer_num from orders where
        orders.ship_weight < 50)

Estimated Cost: 4
Estimated # of Rows Returned: 8

  1) informix.orders: INDEX PATH

    (1) Index Keys: customer_num   (Serial, fragments: ALL)
        Lower Index Filter: informix.orders.customer_num = ANY

    Subquery:
    ---------
    Estimated Cost: 2
    Estimated # of Rows Returned: 8
    (Temp Table For Subquery)

      1) informix.orders: SEQUENTIAL SCAN

            Filters: informix.orders.ship_weight < 50.00
```

The next example is based on the following SQL statements, which include a DELETE operation:

```
DATABASE stores_demo;
SET EXPLAIN ON;
DELETE FROM catalog WHERE stock_num IN
 (SELECT stock.stock_num FROM stock, catalog WHERE
        stock.stock_num = catalog.stock_num
        AND stock.unit_price < 50);
CLOSE DATABASE;
```

Below is the resulting output:

```
QUERY:
------
DELETE FROM catalog WHERE stock_num IN
 (SELECTstock.stock_num from stock, catalog
    WHERE stock.stock_num = catalog.stock_num
    AND stock.unit_price < 50);

Estimated Cost: 19
Estimated # of Rows Returned: 37

  1) ajay.catalog: INDEX PATH

    (1) Index Keys: stock_num manu_code   (Serial, fragments: ALL)
        Lower Index Filter: ajay.catalog.stock_num = ANY

    Subquery:
```

```
    ---------
    Estimated Cost: 12
    Estimated # of Rows Returned: 44
    (Temp Table For Subquery)

      1) ajay.stock: SEQUENTIAL SCAN

            Filters: ajay.stock.unit_price < $50.00

      2) ajay.catalog: INDEX PATH

        (1) Index Keys: stock_num manu_code
            (Key-Only)  (Serial, fragments: ALL)
            Lower Index Filter:
            ajay.stock.stock_num = ajay.catalog.stock_num

NESTED LOOP JOIN
```

The next example, through sqlexplain output, shows how optimizer uses a visible index and ignores an invisible index. tab1 is loaded with unique 1,000,000 rows and has a visible index created.

```
QUERY:
------
select count(*) from tab1 where col1 = 200

Estimated Cost: 1
Estimated # of Rows Returned: 1

  1) uaymak1.tab1: INDEX PATH

    (1) Index Keys: col1 (Key-Only) Serial, fragments: ALL)
        Lower Index Filter: uaymak1.tab1.col1 = 200
```

Now the index is made invisible with the set indexes statement and the query is rerun. It is seen that optimizer ignores the index.

```
QUERY:
------
select count(*) from tab1 where col1 = 200

     Estimated Cost: 35954
     Estimated # of Rows Returned: 1

1) uaymak1.tab1: SEQUENTIAL SCAN
   Filters: uaymak1.tab1.col1 = 200
```

External Table Operations in SET EXPLAIN Output

The Query Statistics section of SET EXPLAIN output provides information on operations that are loading data from or unloading data to an external table.

The following codes in the Query Statistics section of the SET EXPLAIN output file provides information on external tables:

- `xlcnv` identifies an operation that is loading data from an external table and inserting the data into a base table. Here `x` = external table, `l` = loading, and `cnv` = converter
- `xucnv` identifies an operation that is reading data from the base table and writing to the file that the external table is pointing to. Here `x` = external table, `u` = unloading, and `cnv` = converter

Example

Examples

The following example shows a query in which an operating is loading data from an external table and inserting the data into a base table:

```
QUERY: (OPTIMIZATION TIMESTAMP: 11-11-2009 12:55:20)
------
insert into items select * from ext_items

Estimated Cost: 5
Estimated # of Rows Returned: 68

  1) informix.ext_items: SEQUENTIAL SCAN

Query statistics:
-----------------
  Table map :
  ----------------------------
  Internal name     Table name
  ----------------------------
  t1                items

  type     it_count   time
  ----------------------------
  xlread   1          00:00.00

  type     it_count   time
  ----------------------------
  xlcnv    67         00:00.00

  type     table  rows_ins   time
  -----------------------------------
  insert   t1     67         00:00.00
```

The following example shows a query in which an operating is loading data from an external table and inserting the data into a base table:

```
QUERY: (OPTIMIZATION TIMESTAMP: 11-11-2009 12:47:55)
------
select * from orders into external ord_ext
using (datafiles ('disk:/tmp/ord'))



Estimated Cost: 2
Estimated # of Rows Returned: 23

  1) informix.orders: SEQUENTIAL SCAN
```

```
Query statistics:
-----------------

  Table map :
  ---------------------------
  Internal name     Table name
  ---------------------------
  t1                orders

  type     table  rows_prod  est_rows  rows_scan  time       est_cost
  ------------------------------------------------------------------
  scan     t1     23         23        23         00:00.00   3

  type     it_count   time
  ---------------------------
  xucnv    23         00:00.00

  type     it_count   time
  ---------------------------
  xuwrite  23         00:00.00
```

SET INDEXES statement

Use the SET INDEXES statement to enable or disable a user-defined index, to change the filtering mode of a unique index, or to make an index visible or invisible.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_1094_Connect_42_setidx201) unique_1094_Connect_42_setidx201(explicit id unique_1094_Connect_42_setidx203) unique_1094_Connect_42_setidx203 **SET INDEXES** { *index* | **FOR***table* } { { **ENABLED** | **DISABLED** } | { **VISIBLE** | **INVISIBLE** } | [] **FILTERING** { **WITH ERROR** | **WITHOUT ERROR** } [] }

Element	Description	Restrictions	Syntax
index	Index to be enabled, disabled, or changed in its filtering mode	Must exist	Identifier on page 1231
table	Table whose indexes are all to be enabled, disabled, or changed in their filtering mode	Must exist	Identifier on page 1231

Usage

You can use this statement to enable or disable a specific index or a list of indexes. You can also use the *table* option to enable or disable all of the user-defined indexes on a table without specifying their individual identifiers. For example, the next two examples respectively disable and enable all of the indexes on the **cust_calls** table:

```
SET INDEXES FOR cust_calls DISABLED;
SET INDEXES FOR cust_calls ENABLED;
```

This simple syntax can be convenient in operations where you intend to LOAD or TRUNCATE all the data in a table, or to consolidate the free space in a table.

VISIBLE and *INVISIBLE* options are used to make a user-created index on a table visible or invisible. An index that is marked *invisible*, is maintained but ignored by the optimizer. By default a newly created index is VISIBLE.

As shown in the example given below, three indexes idx1, idx2 and idx3 are created on tab1. idx1 and idx2 will be created VISIBLE. Although idx1 is created without the VISIBLE keyword, since a newly created index is visible by default, idx1 will be created visible. idx3 will be created INVISIBLE.

```
CREATE INDEX idx1 on tab1(col1);
CREATE INDEX idx2 on tab1(col2) VISIBLE;
CREATE INDEX idx3 on tab1(col3) INVISIBLE;
```

Using SET INDEXES, idx1 and idx2 will be made INVISIBLE and idx3 will be made visible as shown in this example:

```
SET INDEXES idx1 INVISIBLE;
SET INDEXES idx2 INVISIBLE;
SET INDEXES idx3 VISIBLE;
```

Using the *table* option, you can make all of the user-defined indexes on a table visible/invisible without specifying their individual identifiers. e.g. all the indexes of tab1 can be made visible as shown in this example:

```
SET INDEXES FOR tab1 VISIBLE;
```

Explicitly-defined and implicitly-defined indexes

The SET INDEXES statement operates on indexes that the CREATE INDEX statement created explicitly. It is not useful, however, with system-defined indexes that PRIMARY KEY or FOREIGN KEY constraint definitions create implicitly. The SET INDEXES statement cannot specify system-generated names that begin with the blank (ASCII 32) character, even if your database has the **DELIMIDENT** environment variable set to support double quotation marks as delimiters for database object identifiers.

To enable or disable implicitly-defined indexes, use instead the SET CONSTRAINTS statement, whose FOR *table* option can reference system-generated constraints implicitly, as in the following examples:

```
SET CONSTRAINTS FOR cust_calls DISABLED;
SET CONSTRAINTS FOR cust_calls ENABLED;
```

To disable all the explicitly-defined and implicitly-defined indexes of a table, use the FOR *table* options of both the SET INDEXES and SET CONSTRAINTS statements, as in the following examples:

```
SET INDEXES FOR cust_calls DISABLED;
SET CONSTRAINTS FOR cust_calls DISABLED;
```

You can similarly enable all the explicitly-defined and implicitly-defined indexes of a table, without referencing the system-generated names of the implicitly-defined indexes, by substituting `ENABLED` for `DISABLED` in the examples above.

The SET INDEXES statement is a special case of the SET Database Object Mode statement. The SET Database Object Mode statement can also enable or disable a trigger or constraint, or can change the filtering mode of constraints and unique indexes.

For the complete syntax and semantics of the SET INDEXES statement, see SET Database Object Mode statement on page 784.

Do not confuse the SET INDEXES statement with the SET INDEX statement, which was supported in releases earlier than Version 9.40. The Informix® database server ignores the SET INDEX statement in current releases.

Restrictions on Secondary Servers

In cluster environments, the SET INDEXES statement is not supported on updatable secondary servers. (More generally, session-level index, trigger, and constraint modes that the SET Database Object Mode statement specifies are not redirected for UPDATE operations on table objects in databases of secondary servers.)

SET ISOLATION statement

Use the SET ISOLATION statement to define the degree of concurrency among processes that attempt to access the same rows simultaneously.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
SET ISOLATION [TO] { REPEATABLE READ | { COMMITTED READ [LAST COMMITTED]  | CURSOR STABILITY | DIRTY READ [ WITH WARNING ] }
[ RETAIN UPDATE LOCKS ] }
```

Usage

The SET ISOLATION statement is the Informix® extension to the ANSI SQL-92 standard. The SET ISOLATION statement can change the enduring isolation level for the session. If you want to set isolation levels through an ANSI-compliant statement, use the SET TRANSACTION statement instead. For a comparison of these two statements, see SET TRANSACTION statement on page 837.

The TO keyword is optional, and has no effect.

SET ISOLATION provides the same functionality as the ISO/ANSI-compliant SET TRANSACTION statement for isolation levels of DIRTY READ (called UNCOMMITTED in SET TRANSACTION), COMMITTED READ, and REPEATABLE READ (called SERIALIZABLE in SET TRANSACTION).

The database *isolation_level* affects read concurrency when rows are retrieved from the database. The isolation level specifies the phenomena that can occur during execution of concurrent SQL transactions. The following phenomena are possible:

- **Dirty Read**. SQL transaction T1 modifies a row. SQL transaction T2 then reads that row before T1 performs a COMMIT. If T1 then performs a ROLLBACK, T2 will have read a row that was never committed, and therefore can be considered never to have existed.
- **Non-Repeatable Read**. SQL transaction T1 reads a row. SQL transaction T2 then modifies or deletes that row and performs a COMMIT. If T1 then attempts to reread that row, T1 might receive the modified value or discover that the row has been deleted.

- **Phantom Row**. SQL transaction T1 reads the set of rows N that satisfy some search condition. SQL transaction T2 then executes SQL statements that generate one or more new rows that satisfy the search condition used by SQL transaction T1. If T1 then repeats the original read with the same search condition, T1 receives a different set of rows.

The database server uses shared locks to support different levels of isolation among processes attempting to access data.

The update or delete process always acquires an exclusive lock on the row that is being modified. The level of isolation does not interfere with rows that you are updating or deleting. If another process attempts to update or delete rows that you are reading with an isolation level of Repeatable Read, that process is denied access to those rows.

In , cursors that are open when SET ISOLATION executes might or might not use the new isolation level when rows are retrieved. Any isolation level that was set from the time the cursor was opened until the application fetches a row might be in effect. The database server might have read rows into internal buffers and internal temporary tables using the isolation level that was in effect at that time. To ensure consistency and reproducible results, close any open cursors before you execute the SET ISOLATION statement.

You can issue the SET ISOLATION statement from a client computer only after a database is opened.

Complete-Connection Level Settings

The SET ISOLATION statement supports *complete-connection level* settings. This means that values in the local session environment at the time of connection are propagated to all new or resumed transactions. These can include the following types of transactions:

- transactions within the local database,
- distributed transactions across databases of the same server instance,
- distributed transactions across databases of two or more database server instances,
- global transactions with XA-compliant data sources that are registered in the local database.

If you change the isolation level within a transaction, the new value is propagated back to the local environment and also to all subsequent new or resumed transactions.

Informix® Isolation Levels

The following definitions explain the critical characteristics of each isolation level, from the lowest level of isolation to the highest.

Using the Dirty Read Isolation Level

Use the Dirty Read option to copy rows from the database whether or not there are locks on them. The program that fetches a row places no locks and it respects none. Dirty Read is the only isolation level available to databases that do not implement transaction logging.

This isolation level is most appropriate for static tables that are used for queries of tables where data is not being modified, because it provides no isolation. With Dirty Read, the program might return an uncommitted row that was inserted or

modified within a transaction that has subsequently rolled back, or a *phantom row* that was not visible when you first read the query set, but that materializes in the query set before a subsequent read within the same transaction. (Only the Repeatable Read isolation level prevents access to phantom rows. Only Dirty Read provides access to uncommitted rows from concurrent transactions that might subsequently be rolled back.)

The optional WITH WARNING keywords instruct the database server to issue a warning when DML operations that use the Dirty Read isolation level might return an uncommitted row or a phantom row. The transaction in the following example uses this isolation level:

```
BEGIN WORK;
SET ISOLATION TO DIRTY READ WITH WARNING;
...
COMMIT WORK;
```

The Dirty Read isolation level is sensitive to the current setting of the USELASTCOMMITTED configuration parameter and of the **USELASTCOMMITTED** session environment variable. For information about the behavior of the Dirty Read isolation level when either of these are set to `DIRTY READ` or to `ALL`, see #unique_858.

When you use High Availability Data Replication, the database server effectively uses Dirty Read isolation on the HDR Secondary Server, regardless of the specified SET ISOLATION or SET TRANSACTION isolation level, unless the UPDATABLE_SECONDARY configuration parameter is enabled. For more information about this topic, see Isolation Levels for Secondary Data Replication Servers on page 817.

Using the Committed Read Isolation Level

Use the Committed Read option to guarantee that every retrieved row is committed in the table at the time that the row is retrieved. This option does not place a lock on the fetched row. Committed Read is the default level of isolation in a database with logging that is not ANSI compliant.

Committed Read is appropriate when each row is processed as an independent unit, without reference to other rows in the same table or in other tables.

Using the Cursor Stability Isolation Level

Use the Cursor Stability option to place a shared lock on the fetched row, which is released when you fetch another row or close the cursor. Another process can also place a shared lock on the same row, but no process can acquire an exclusive lock to modify data in the row. Such row stability is important when the program updates another table based on the data it reads from the row.

If you set the isolation level to Cursor Stability, but you are not using a transaction, the Cursor Stability acts like the Committed Read isolation level.

Using the Repeatable Read Isolation Level

Use the Repeatable Read option to place a shared lock on every row that is selected during the transaction. Another process can also place a shared lock on a selected row, but no other process can modify any selected row during your transaction, nor insert a row that meets the search criteria of your query during your transaction. If you repeat the query during the

transaction, you reread the same information. The shared locks are released only when the transaction commits or rolls back. Repeatable Read is the default isolation level in an ANSI-compliant database.

Repeatable Read isolation places the largest number of locks and holds them the longest. Therefore, it is the level that reduces concurrency the most.

Default Isolation Levels

The default isolation level for a particular database is established when you create the database according to database type. The following list describes the default isolation level for each database type.

Isolation Level

Database Type

Dirty Read

Default level in a database without logging

Committed Read

Default level in a logged database that is not ANSI compliant

Repeatable Read

Default level in an ANSI-compliant database

The default level remains in effect until you issue a SET ISOLATION statement. After a SET ISOLATION statement executes, the new isolation level remains in effect until one of the following events occurs:

- You enter another SET ISOLATION statement.
- You open another database that has a default isolation level different from the level that your last SET ISOLATION statement specified.
- The program ends.

For a Informix® database that is not ANSI-compliant, unless you explicitly set the USELASTCOMMITTED configuration parameter, the LAST COMMITTED feature is not in effect for the default isolation levels. The SET ENVIRONMENT statement or the SET ISOLATION statement can override this default and enable LAST COMMITTED for the current session.

Using the RETAIN UPDATE LOCKS Option

Use the RETAIN UPDATE LOCKS option to affect the behavior of the database server when it handles a SELECT ... FOR UPDATE statement.

In a database with the isolation level set to Dirty Read, Committed Read, or Cursor Stability, the database server places an update lock on a fetched row of a SELECT ... FOR UPDATE statement. When you turn on the RETAIN UPDATE LOCKS option, the database server retains the update lock until the end of the transaction rather than releasing it at the next subsequent FETCH or when the cursor is closed. This option prevents other users from placing an exclusive lock on the updated row before the current user reaches the end of the transaction.

You can use this option to achieve the same locking effects but avoid the overhead of dummy updates or the repeatable read isolation level.

You can turn this option on or off at any time during the current session.

You can turn the option off by resetting the isolation level without using the RETAIN UPDATE LOCKS keywords, as in the following example.

```
BEGIN WORK;
SET ISOLATION TO
   COMMITTED READ LAST COMMITTED RETAIN UPDATE LOCKS;
...
COMMIT WORK;
BEGIN WORK;
SET ISOLATION TO COMMITTED READ LAST COMMITTED ;
...
COMMIT WORK;
```

Controlling Update Locks through the Session Environment

Another way to disable RETAIN UPDATE LOCKS behavior is to execute this SQL statement:

```
SET ENVIRONMENT RETAINUPDATELOCKS 'NONE';
```

This disables the RETAIN UPDATE LOCKS clause for the current transaction, and for any subsequent transactions of the same session, by resetting the RETAINUPDATELOCKS session environment variable.

The SET ENVIRONMENT RETAINUPDATELOCKS statement can also make the retention of update locks the default behavior for either the Committed Read, Cursor Stability, or Dirty Read isolation levels, or for all of these isolation levels, regardless of whether the SET ISOLATION statement includes the RETAIN UPDATE LOCKS clause.

For more information on update locks, see RETAINUPDATELOCKS session environment option and Locking Considerations on page 872.

Turning the Option OFF During a Transaction

If you set the RETAIN UPDATE LOCKS option to OFF after a transaction has begun, but before the transaction has been committed or rolled back, several update locks might still exist.

Switching OFF the feature does not directly release any update lock. When you turn this option off, the database server reverts to normal behavior for the three isolation levels. That is, a FETCH statement releases the update lock placed on a row by the immediately preceding FETCH statement, and a closed cursor releases the update lock on the current row.

Update locks placed by earlier FETCH statements are not released unless multiple update cursors are present within the same transaction. In this case, a subsequent FETCH could also release older update locks of other cursors.

Effects of Isolation Levels

You cannot set the transaction isolation level in a database that does not have logging. Every retrieval in such a database occurs as a Dirty Read.

The data retrieved from a BYTE or TEXT column can vary, depending on the transaction isolation level. Under Dirty Read or Committed Read levels of isolation, a process can read a BYTE or TEXT column that is either deleted (if the delete is not yet committed) or in the process of being deleted. Under these isolation levels, deleted data is readable under certain conditions. For information about these conditions, see the *HCL® Informix® Administrator's Guide*.

When you use , as you use higher levels of isolation, lock conflicts occur more frequently. For example, if you use Cursor Stability, more lock conflicts occur than if you use Committed Read.

Using a scroll cursor in an ESQL/C transaction, you can force consistency between your temporary table and the database table either by setting the level to Repeatable Read or by locking the entire table during the transaction.

If you use a scroll cursor WITH HOLD in a transaction, you cannot force consistency between your temporary table and the database table. A table-level lock or locks that are set by Repeatable Read are released when the transaction is completed, but the scroll cursor with hold remains open beyond the end of the transaction. You can modify released rows as soon as the transaction ends, but retrieved data in the temporary table might be inconsistent with the actual data.

Attention: Do not use nonlogging tables within a transaction. If you need to use a nonlogging table within a transaction, either set the isolation level to Repeatable Read or else lock the table in Exclusive mode to prevent concurrency problems.

Isolation Levels for Secondary Data Replication Servers

If the UPDATABLE_SECONDARY configuration parameter is disabled (by being unset or by being set to zero), a secondary data replication server is read-only. In this case, only the Dirty Read or Read Uncommitted transaction isolation levels are available on High-Availability Data Replication (HDR) and Remote Standalone Secondary (RSS) servers.

If the UPDATABLE_SECONDARY parameter is enabled (by being set to a valid number of connections greater than zero), a secondary data replication server can support the Read Committed, Committed Read, or Committed Read Last Committed transaction isolation level, with or without the USELASTCOMMITTED session environment variable of the SET ENVIRONMENT statement. Only the DELETE, INSERT, UPDATE, and MERGE statements of SQL(and the **dbexport** utility, if the STOP_APPLY, USELASTCOMMITTED, and UPDATABLE_SECONDARY configuration parameters are set) can support write operations on an updatable secondary server.

Shared Disk Secondary (SDS) servers, however, can support the Read Committed, Committed Read, Committed Read Last Committed isolation levels, regardless of their UPDATABLE_SECONDARY setting. For more information about the UPDATABLE_SECONDARY configuration parameter, see the HCL Informix® Administrator's Reference.

SET LOCK MODE statement

Use the SET LOCK MODE statement to define how the database server handles a process that tries to access a locked row or table.

This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
SET LOCK MODE TO { NOT WAIT | WAIT [ seconds ] }
```

Element	Description	Restrictions	Syntax
seconds	Maximum number of seconds that a process waits for a lock to be released before issuing an error	Valid only if shorter than system default	Literal Number on page 1198

Usage

This statement can direct the response of the database server in the following ways when a process tries to access a locked row or table.

Lock Mode

Effect

NOT WAIT

Database server ends the operation immediately and returns an error code. This condition is the default.

WAIT

Database server suspends the process until the lock releases.

WAIT *seconds*

Database server suspends the process until the lock releases or until the waiting period ends. If the lock remains after the waiting period, the operation ends and an error code is returned.

For a description of the two distinct meanings of the term *lock mode* in this document, see Locking Granularity in the related concepts section.

To avoid waiting in operations that attempt to read rows on which concurrent sessions hold exclusive row-level locks, you can also use the LAST COMMITTED feature, either by setting it explicitly in the SET ISOLATION COMMITTED READ statement, or by setting the USELASTCOMMITTED configuration parameter or the USELASTCOMMITTED session environment option.

Examples

In the following example, the user specifies that if the process requests a locked row, the operation should end immediately and an error code should be returned:

```
SET LOCK MODE TO NOT WAIT;
```

In the following example, the user specifies that the process should be suspended until the lock is released:

```
SET LOCK MODE TO WAIT;
```

The next example sets an upper limit of 17 seconds on the length of any wait:

```
SET LOCK MODE TO WAIT 17;
```

WAIT Clause

The WAIT clause causes the database server to suspend the process until the lock is released or until a specified number of seconds have passed without the lock being released.

The database server protects against the possibility of a deadlock when you request the WAIT option. Before the database server suspends a process, it checks whether suspending the process could create a deadlock. If the database server discovers that a deadlock could occur, it ends the operation (overruling your instruction to wait) and returns an error code. In the case of either a suspected or actual deadlock, the database server returns an error.

Cautiously use the unlimited waiting period that was created when you specify the WAIT option without *seconds*. If you do not specify an upper limit, and the process that placed the lock somehow fails to release it, suspended processes could wait indefinitely. Because a true deadlock situation does not exist, the database server does not take corrective action.

In a network environment, the DBA uses the ONCONFIG parameter DEADLOCK_TIMEOUT to establish a default value for *seconds*. If you use a SET LOCK MODE statement to set an upper limit, your value applies only when your waiting period is shorter than the system default.

Complete-Connection Level Settings

The SET LOCK MODE statement supports *complete-connection level* settings. This means that values in the local session environment at the time of connection are propagated to all new or resumed transactions. These can include the following types of transactions:

- transactions within the local database,
- distributed transactions across databases of the same server instance,
- distributed transactions across databases of two or more database server instances,
- global transactions with XA-compliant data sources that are registered in the local database.

If you change the lock mode setting within a transaction, the new value is propagated back to the local environment and also to all subsequent new or resumed transactions.

In releases of Informix® earlier than 9.40.UC8, the SET LOCK MODE statement did not support complete-connection level settings. The process waited for the specified number of seconds only if you acquired locks within the current database server and a remote database server within the same transaction.

SET LOG statement

Use the SET LOG statement to change your database logging mode from buffered transaction logging to unbuffered transaction logging or vice versa.

This statement is an extension to the ANSI/ISO standard for SQL. Unlike most extensions, the SET LOG statement is not valid in an ANSI-compliant database.

Syntax

```
SET [ BUFFERED ] LOG
```

Usage

You activate transaction logging when you create a database or add logging to an existing database. These transaction logs can be buffered or unbuffered.

Buffered logging is a type of logging that holds transactions in a memory buffer until the buffer is full, regardless of when the transaction is committed or rolled back. The database server provides this option to speed up operations by reducing the number of disk writes.



Attention: You gain a marginal increase in efficiency with buffered logging, but you incur some risk. In the event of a system failure, the database server cannot recover any completed transactions in the memory buffer that were not written to disk.

The SET LOG statement in the following example changes the transaction logging mode to buffered logging:

```
SET BUFFERED LOG;
```

Unbuffered logging is a type of logging that does not hold transactions in a memory buffer. As soon as a transaction ends, the database server writes the transaction to disk. If a system failure occurs when you are using unbuffered logging, you recover all completed transactions, but not those still in the buffer. The default condition for transaction logs is unbuffered logging.

The SET LOG statement in the following example changes the transaction logging mode to unbuffered logging:

```
SET LOG;
```

The SET LOG statement redefines the mode for the current session only. The default mode, which the database administrator sets with the **ondblog** utility, remains unchanged.

The buffering option does not affect retrievals from external tables. For distributed queries, a database with logging can retrieve only from databases with logging, but it makes no difference whether the databases use buffered or unbuffered logging.

An ANSI-compliant database cannot use buffered logging.

You cannot change the logging mode of ANSI-compliant databases. If you created a database with the WITH LOG MODE ANSI keywords, you cannot later use the SET LOG statement to change the logging mode to buffered or unbuffered transaction logging.

SET OPTIMIZATION statement

Use the SET OPTIMIZATION statement to specify how much time the query execution optimizer spends developing a query plan or specifying optimization goals. The SET OPTIMIZATION statement is an extension to the ANSI/ISO standard for SQL

When you use **DB-Access** with Informix®, the ENVIRONMENT options to the SET OPTIMIZATION statement can define a general optimization environment for all statements in the current session.

Syntax

(explicit id unique_1170_Connect_42_sop002) unique_1170_Connect_42_sop002(explicit id unique_1170_Connect_42_sop003)

unique_1170_Connect_42_sop003 **SET OPTIMIZATION** { { **HIGH** | **LOW** } | { { **FIRST_ROWS** | **ALL_ROWS** } } | [<ENVIRONMENT Options>] }

Usage

You can execute a SET OPTIMIZATION statement at any time. The specified optimization level carries across databases on the current database server. The option that you specify remains in effect until you issue another SET OPTIMIZATION statement or until the program ends. The default database server optimization level for the amount of time that the query optimizer spends determining the query plan is HIGH.

On Informix®, the default optimization goal is ALL_ROWS. Although you can set only one option at a time, you can issue two SET OPTIMIZATION statements: one that specifies the time the optimizer spends to determine the query plan and one that specifies the optimization goal of the query.

Similarly, you can issue multiple SET OPTIMIZATION statements that include the ENVIRONMENT Options clause to specify a session environment for optimizing queries. In data warehousing applications, an appropriate optimizer environment can improve the performance of join queries of tables in a star schema. Optimizer environment settings persist until another SET OPTIMIZATION ENVIRONMENT statement overrides them, or until the session ends. For more information, see the ENVIRONMENT Options on page 823 topic.

Examples

The following example shows optimization across a network. The **central** database (on the **midstate** database server) is to have LOW optimization; the **western** database (on the **rockies** database server) is to have HIGH optimization.

```
CONNECT TO 'central@midstate';
SET OPTIMIZATION LOW;
SELECT * FROM customer;
CLOSE DATABASE;
CONNECT TO 'western@rockies';
SET OPTIMIZATION HIGH;
SELECT * FROM customer;
CLOSE DATABASE;
CONNECT TO 'wyoming@rockies';
SELECT * FROM customer;
```

Here the **wyoming** database is to have HIGH optimization because it resides on the same database server as the **western** database. The code does not need to re-specify the optimization level for the **wyoming** database because the **wyoming** database resides on the **rockies** database server like the **western** database.

The following example directs the Informix® optimizer to use the most time to determine a query plan, and to then return the first rows of the result as soon as possible:

```
SET OPTIMIZATION LOW;
SET OPTIMIZATION FIRST_ROWS;
```

```
SELECT lname, fname, bonus
   FROM sales_emp, sales
   WHERE sales.empid = sales_emp.empid AND bonus > 5,000
      ORDER BY bonus DESC;
```

HIGH and LOW Options

The HIGH and LOW options determine how much time the query optimizer spends to determine the query plan:

- HIGH

  This option directs the optimizer to use a sophisticated cost-based algorithm that examines all reasonable query-plan choices and selects the best overall alternative.

  For large joins, this algorithm can incur more overhead than you desire. In extreme cases, you can run out of memory.

- LOW

  This option directs the optimizer to use a less sophisticated but faster to design optimization algorithm, based on the lowest-cost path at each stage. This algorithm eliminates unlikely join strategies during the early stages of optimization and reduces the time and resources spent during optimization.

  When you specify the LOW level of optimization, the database server might not select the optimal strategy because that strategy was eliminated from consideration during the early stages of the algorithm.

FIRST_ROWS and ALL_ROWS Options

The `FIRST_ROWS` and `ALL_ROWS` keyword options identify two different query optimization goals:

- **`FIRST_ROWS`**

  This option directs the optimizer to choose the query plan that returns the first qualifying record as soon as possible, ignoring plans that would sort records or create a hash table.

- **`ALL_ROWS`**

  This option directs the optimizer to choose the query plan that returns all the qualifying records as quickly as possible.

Inline optimizer directives

Rather than choosing either of these SET OPTIMIZATION statement options, you might instead specify an optimization-goal directive for an individual query as a comment immediately following the SELECT keyword that begins the query or subquery. For more information about the syntax and options for inline optimizer directives for queries in DELETE, SELECT, or UPDATE statements, see Optimizer Directives on page 1246.

External optimizer directives

Besides the query optimizer directives that the SET OPTIMIZATION statement can specify for queries in the current session, or the inline optimizer directives that can follow the SELECT keyword, the database server also supports a third format for influencing the choice of execution path by the query optimizer.

The DBA or user **informix** can execute the SAVE EXTERNAL DIRECTIVES statement to register *external optimizer directives*, also called *external directives*, in the sysdirectives table of the system catalog. If support for external directives has been enabled, the Informix® database server applies these directives automatically to queries that match a specified SELECT statement. The `ACTIVE`, `INACTIVE`, or `TEST ONLY` keyword options of SAVE EXTERNAL DIRECTIVES statements respectively enable, disable, or restrict the scope of an external directive.

Setting the EXT_DIRECTIVES configuration parameter in the ONCONFIG file to `1` or `2`, and setting the IFX_EXTDIRECTIVES client-side environment variable to `1` enables support for external directives.

Independent of the EXT_DIRECTIVES configuration setting or the IFX_EXTDIRECTIVES client-side settings, the SET ENVIRONMENT EXTDIRECTIVES setting of `'1'`, `on`, or `ON` for the session environment enables external optimizer directives in the current user session, if any active external directives are registered in the sysdirectives table.

The SET EXPLAIN output file displays whether external directives are in effect for a query.

For more information about external optimizer directives, see Enabling or disabling external directives for a session on page 686.

Optimizing SPL Routines

For SPL routines that remain unchanged or change only slightly, you might want to set the SET OPTIMIZATION statement to HIGH when you create the SPL routine. This step stores the best query plans for the SPL routine. Then execute a SET OPTIMIZATION LOW statement before you execute the SPL routine. The SPL routine then uses the optimal query plans and runs at the more cost-effective rate.

ENVIRONMENT Options

Use the ENVIRONMENT Options clause of the SET OPTIMIZATION statement to define attributes of the optimization environment in the current session. These attributes persist until the session ends, or until another SET OPTIMIZATION ENVIRONMENT statement resets an optimization directive.

For some data warehousing applications, session environment settings that you specify in this clause can improve the performance of queries that join fact tables with dimension tables, in databases where the primary key of each dimension table corresponds to a foreign key of the fact table.

The **DB-Access** utility of Informix® supports the ENVIRONMENT Options clause of the SET OPTIMIZATION statement.

Syntax

```
ENVIRONMENT { STAR_JOIN ' { ENABLED | DISABLED | FORCED } ' | USE_INVISIBLE_INDEXES ' { { TRUE | YES | 1 } | { FALSE | NO |
0 } } ' | { FACT | AVOID_FACT | NON_DIM } { ' table ' | { DEFAULT | '' } } }
```

Element	Description	Restrictions	Syntax
table	Table, view, or synonym	Must exist in the database	Identifier on page 1231

Usage

This syntax diagram is simplified, and does not show the double (") quotation mark option for delimiters enclosing the list of one or more *table* objects, or enclosing the `DISABLED`, `ENABLED`, or `FORCED` keyword that you specify as the optimization environment setting.

Important: Do not use quotation marks to delimit the `STAR_JOIN`, `FACT`, `AVOID_FACT`, `NON_DIM`, or `DEFAULT` keywords. In the ENVIRONMENT Options clause, the `DEFAULT` keyword and the empty string (`''` or `""`) are equivalent.

The following table lists each ENVIRONMENT option, and describes their effect on whether the query optimizer considers star-join execution plans.

Keywords	Effect	Optimizer Action
`STAR_JOIN`	The `'ENABLED'` setting turns on (and `'DISABLED'` turns off) star-join support for the current session. The `'FORCED'` setting favors a star-join execution path, when possible, for all queries	For `'ENABLED'`, the optimizer considers the possibility of a star-join execution plan. For `'FORCED'`, a star-join plan will be chosen, if available. For `'DISABLED'`, star-join is not considered.
`FACT`	Identifies tables that correspond to fact tables in a star schema. If an `AVOID_FACT` table is also specified in the same session as `FACT`, then `FACT` takes precedence. `DEFAULT` or an empty string turns off the `FACT` directive for the session.	Only tables in the `FACT` list are considered as fact tables in star-join optimization. Multiple tables can be listed as `FACT`.
`AVOID_FACT`	Do not use the table (or any table in the list of tables) as a fact table in star-join optimization. `DEFAULT` or an empty string turns off the `AVOID_FACT` directive for the session.	Tables in the `AVOID_FACT` list are not considered as fact tables in star-join optimization. Multiple tables can be listed as `AVOID_FACT`.
`NON_DIM`	Identifies tables that do not correspond to dimension tables in a star schema. `DEFAULT` or an empty string turns off the `NON_DIM` directive for the session.	Tables in the `NON_DIM` list are not considered as dimension tables in star-join optimization. Multiple tables can be listed as `NON_DIM`.
`USE_INVISIBLE_INDEXES`	The `'YES/TRUE/1'` setting turns off invisibility of indexes for that session. The `'NO/FALSE/0'` setting turns on invisibility of indexes for that session .	For `'YES/TRUE/1'`, optimizer considers the invisible index for that session. For `'NO/FALSE/0'`, optimizer does not consider the invisible index for that session.

Unless you explicitly set the `FACT`, `AVOID_FACT`, or `NON_DIM` optimizer directive to `DEFAULT`, a valid SET OPTIMIZATION ENVIRONMENT statement must include exactly one quoted string that defines an optimization environment setting. When the setting for the directive that follows the `ENVIRONMENT` keyword is not `DEFAULT`, a pair of single (`'`) or double (`"`) quotation mark symbols must delimit the last specification:

- Either the `'ENABLED'`, `'DISABLED'`, or `'FORCED'` keyword that follows the STAR_JOIN directive,
- Either the `'YES/TRUE/1'` or `'NO/FALSE/0'` keyword that follows the USE_INVISIBLE_INDEXES directive,
- or the comma-separated list of one or more *table* identifiers that specifies the setting of an optimizer environment attribute.
- If the setting is the empty string or the `DEFAULT` keyword, the `FACT`, `AVOID_FACT`, or `NON_DIM` optimizer directive that follows the `ENVIRONMENT` keyword does not affect the query optimizer during subsequent queries in the same session. You can use this option to disable a previously specified `FACT`, `AVOID_FACT`, or `NON_DIM` optimizer directive.

The following restrictions apply to blank spaces within the SET OPTIMIZATION ENVIRONMENT statement:

- If a comma-separated list of more than one *table* identifier follows the `FACT`, `AVOID_FACT`, or `NON_DIM` keywords, do not include blank spaces between any of the items in the list.
- Similarly, do not include blank characters between the two single (`''`) or two double (`""`) delimiters of the empty string that is a synonym for the `DEFAULT` keyword.

For example, the following statements override any previous directives to prevent the query optimizer from considering any specific tables as fact tables or as dimension tables:

```
SET OPTIMIZATION ENVIRONMENT AVOID_FACT "";
SET OPTIMIZATION ENVIRONMENT NON_DIM '';
```

By associating the `AVOID_FACT`, or `NON_DIM` keywords with an empty set of tables, the statements in the examples above allow any table to be considered as a fact table or as a dimension table in a star-join execution path.

USE_INVISIBLE_INDEXES optimizer directive impacts query optimizer's way of handling only **invisible** indexes i.e. value of USE_INVISIBLE_INDEXES decides if an invisible index will be considered or ignored during a query execution. The variable only changes the index visibility temporarily within the session.

The FALSE value has no impact unless within the same session the variable was set to TRUE to make the optimizer consider the index.

```
SET INDEXES idx1 INVISIBLE;
```

- Above statement will result in optimizer ignoring the index while running a query.

```
SET OPTIMIZATION ENVIRONMENT USE_INVISIBLE_INDEXES “TRUE”;
```

- After this statement, i.e. when USE_INVISIBLE_INDEXES is TRUE/YES/1, optimizer will consider the index while running a query.

```
SET OPTIMIZATION ENVIRONMENT USE_INVISIBLE_INDEXES “FALSE”;
```

- As a result of this statement i.e. when USE_INVISIBLE_INDEXES is FALSE/NO/0, optimizer will ignore the index while running a query.

Configuring optimization at connection time

The DBA can use an **sysdbopen( )** routine to define an optimization environment that takes effect when the user connects to the database. For example, to specify an optimizer environment that always favors a star-join execution plan, the **sysdbopen( )** routine should include these SQL statements:

```
SET OPTIMIZATION ENVIRONMENT STAR_JOIN 'FORCED';
SET OPTIMIZATION ENVIRONMENT FACT 'table1,table2, ... tableN';
```

In the SET OPTIMIZATION ENVIRONMENT FACT statement, the tables listed after the `FACT` keyword should all be fact tables.

Additional methods for enabling join directives

The SET OPTIMIZATION ENVIRONMENT statement is one of several ways to specify FACT optimizer directives:

- Directives within the SELECT statement, embedded comment-like. The scope is the SQL statement.
- The SET OPTIMIZATION ENVIRONMENT FACT statement. The scope is the session.
- The SAVE EXTERNAL DIRECTIVES statement. The scope is the database. This can define FACT directives and save them in the **sysdirectives** system catalog table.

Three mechanisms exist for enabling the query optimizer to consider external directives in the **sysdirectives** system catalog table, including any FACT directives. This is the (approximately) ascending order of precedence:

- The **EXT_DIRECTIVES** configuration parameter. If this is set to `2`, external directives are enabled for the database server, unless the **IFX_EXTDIRECTIVES** environment variable on the client system is set to `0`.
- The **IFX_EXTDIRECTIVES** environment variable. If this is set to `1` on the client system, external optimizer directives are enabled for the database server, unless the **EXT_DIRECTIVES** configuration parameter is set to `0`.
- The SET ENVIRONMENT statement. To enable the query optimizer to consider the external optimizer directives in **sysdirectives** when it chooses an execution path for queries during the current session, the DBA can issue any of these statements:

  ```
  SET ENVIRONMENT EXTDIRECTIVES on;
  SET ENVIRONMENT EXTDIRECTIVES ON;
  SET ENVIRONMENT EXTDIRECTIVES '1';
  SET ENVIRONMENT EXTDIRECTIVES "1";
  ```

  To prevent the query optimizer from considering external optimizer directives in **sysdirectives**, the DBA can issue any of these statements:

  ```
  SET ENVIRONMENT EXTDIRECTIVES off;
  SET ENVIRONMENT EXTDIRECTIVES OFF;
  SET ENVIRONMENT EXTDIRECTIVES '0';
  SET ENVIRONMENT EXTDIRECTIVES "0";
  ```

  These statements override (for the current session only) any conflicting settings of the **EXT_DIRECTIVES** configuration parameter or of the **IFX_EXTDIRECTIVES** environment variable.

For information about the SET ENVIRONMENT EXTDIRECTIVES statement, see EXTDIRECTIVES session environment option.

For a table showing the effects of various combinations of **EXT_DIRECTIVES** settings and **IFX_EXTDIRECTIVES** settings on enabling or disabling external optimizer directives, see Enabling or disabling external directives for a session on page 686.

For information about query optimizer directives that can favor or avoid star-join execution plans, see Star-Join Directives on page 1256.

For information about how to use the built-in **sysdbopen( )** routine to define the session environment at the time of connection for a specified user, for the PUBLIC group, or for a role, see Session Configuration Procedures.

SET PDQPRIORITY statement

The SET PDQPRIORITY statement enables an application to set the query priority level dynamically within a routine. The SET PDQPRIORITY statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
SET PDQPRIORITY { DEFAULT | LOW | OFF | HIGH | resources }
```

Element	Description	Restrictions	Syntax
resources	Integer that specifies the query priority level and the percent of resources to process the query	Can range from -1 to 100. See also Allocating Database Server Resources on page 829.	Literal Number on page 1198

Usage

The SET PDQPRIORITY statement overrides the PDQPRIORITY environment variable (but has lower precedence than the **MAX_PDQPRIORITY** configuration parameter). The scope of SET PDQPRIORITY is local to the routine, and does not affect other routines within the same session. When a routine that issues this statement terminates, the setting reverts to the system default value.

Set PDQ priority to a value less than the quotient of 100 divided by the maximum number of prepared statements. For example, if two prepared statements are active, you should set the PDQ priority to less than `50`.

For example, assume that the DBA sets the **MAX_PDQPRIORITY** parameter to 50. Then a user enters the following SET PDQPRIORITY statement to set the query priority level to 80 percent of resources:

```
SET PDQPRIORITY 80;
```

When it processes the query, the database server uses the **MAX_PDQPRIORITY** value to factor the query priority level set by the user. The database server silently processes the query with a priority level of 40. This priority level represents 50 percent of the 80 percent of resources that the user specifies.

The following keywords are supported by the SET PDQPRIORITY statement.

Keyword

Effect

DEFAULT

Uses the setting of the **PDQPRIORITY** environment variable

LOW

Data values are fetched from fragmented tables in parallel. (In Informix®, when you specify LOW, the database server uses no other forms of parallelism.)

OFF

PDQ is turned off (Informix® only). The database server uses no parallelism. OFF is the default if you use neither the PDQPRIORITY environment variable nor the SET PDQPRIORITY statement.

HIGH

The database server determines an appropriate PDQPRIORITY value, based on factors that include the number of available processors, the fragmentation of the tables being queried, the complexity of the query, and others. IBM® reserves the right to change the performance behavior of queries when HIGH is specified in future releases.

The following SET PDQPRIORITY statements have the same effects:

```
SET PDQPRIORITY DEFAULT;

SET PDQPRIORITY -1;
```

Both replace any prior SET PDQPRIORITY level with the setting of the PDQPRIORITY environment variable. See also, however, the following description of the order of precedence among methods for allocating query priority resources.

Precedence of methods for allocating PDQ priority

For memory available in operations that are affected by PDQPRIORITY, the maximum amount of memory that the database server can allocate is limited by the physical memory available to your system, and by the settings (in ascending order of lowest precedence to highest precedence) of the following environment variables, SQL statements, configuration parameters, and session environment variables:

- The PDQPRIORITY environment variable
- The most recent SET PDQPRIORITY statement of SQL
- The **MAX_PDQPRIORITY** configuration parameter
- The **DS_TOTAL_MEMORY** configuration parameter
- The BOUND_IMPL_PDQ session environment variable, if the IMPLICIT_PDQ session environment variable is enabled in the current session by the SET ENVIRONMENT statement of SQL.

The scope of the SET ENVIRONMENT BOUND_IMPL_PDQ and SET ENVIRONMENT IMPLICIT_PDQ statements is the current session.

When concurrent queries are running, the **DS_MAX_QUERIES** configuration parameter setting can also restrict the amount of PDQ memory available for a new query.

Allocating Database Server Resources

You can specify an integer in the range from `-1` to `100` to indicate a query priority level as the percent of database server resources to process the query. Resources include the amount of memory and the number of processors. The higher the number you specify, the more resources the database server uses.

Use of more resources usually indicates better performance for a given query. Using excessive resources, however, can cause contention for resources and remove resources from other queries, so that degraded performance results. With the *resources* option, the following values are numeric equivalents of the keywords that indicate query priority level.

Value	Equivalent Keyword-Priority Level
`-1`	DEFAULT
`0`	OFF
`1`	LOW

For Informix®, the following statements are equivalent. The first statement uses the keyword LOW to establish a low query-priority level. The second uses a value of `1` in the *resources* parameter to establish a low query-priority level.

```
SET PDQPRIORITY LOW;

SET PDQPRIORITY 1;
```

SET ROLE statement

Use the SET ROLE statement to enable the privileges of a user-defined role. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

SET ROLE { { *role* | *'role'* } | { **NULL** | **NONE** | **DEFAULT** } }

Element	Description	Restrictions	Syntax
role	Name of a role to be enabled	Must already exist in the database and must already have been granted to the user, but cannot be a built-in role. If enclosed between quotation marks, *role* is case sensitive.	Owner name on page 1262;

Usage

Any user who is granted a role can enable the role by using the SET ROLE statement. You can only enable one role at a time. If you execute the SET ROLE statement after a role is already set, the new role replaces the old role as the current role.

The SET ROLE statement returns an error if the user does not currently hold the role, or if the role is a built-in role. (The access privileges held by a built-in role, such as the EXTEND role or the DBSECADM role, are always in effect, and do not require activation by the SET ROLE statement if the user holds that role.)

Users can be granted a default role for the database instance when the DBA issues the GRANT DEFAULT ROLE statement. If no default role exists for the user in the current database, role NULL or NONE is assigned by default. In this context, NULL and NONE are synonyms. Roles NULL and NONE can have no privileges. To set your role to NULL or NONE disables your current role.

When you use SET ROLE to enable a role, you gain the privileges of the role, in addition to the privileges of PUBLIC and your own privileges. If a role is granted to another role that has been assigned to you, you gain the privileges of both roles, in addition to any privileges of PUBLIC and your own privileges.

After SET ROLE executes successfully, the specified role remains effective until the current database is closed or the user executes another SET ROLE statement. Only the user, however, not the role, retains ownership of any database objects, such as tables, that were created during the session.

A role is in scope only within the current database. You cannot use privileges that you acquire from a role to access data in another database. For example, if you have privileges from a role in the database named **acctg**, and you execute a distributed query over the databases named **acctg** and **inventory**, your query cannot access the data in the **inventory** database unless you were also granted appropriate privileges in the **inventory** database. As a security precaution, discretionary access privileges that the user holds only from a role cannot provide access to tables outside the current database through a view or through the action of a trigger.

If your database supports explicit transactions, you must issue the SET ROLE statement outside a transaction. If your database is ANSI-compliant, SET ROLE must be the first statement of a new transaction. If the SET ROLE statement is executed while a transaction is active, an error occurs. For more information about SQL statements that initiate an implicit transaction, see SET SESSION AUTHORIZATION and Transactions on page 833.

If the SET ROLE statement is executed as a part of a trigger or SPL routine, and the owner of the trigger or SPL routine was granted the role with the WITH GRANT OPTION, the role is enabled even if you are not granted the role. For example, this code fragment sets a role and then relinquishes it after a query:

```
EXEC SQL set role engineer;
EXEC SQL select fname, lname, project
      INTO :efname, :elname, :eproject FROM projects
      WHERE project_num > 100 AND lname = 'Larkin';
printf ("%s is working on %s\n", efname, eproject);
EXEC SQL set role NULL;
```

Setting the Default Role

The DBA or the owner of the database can issue the GRANT DEFAULT ROLE statement to assign an existing role as the *default role* to a specified list of users or to PUBLIC. Unlike a non-default role, the default role does not require the SET ROLE statement to enable it. When a user is assigned to the default role, an implicit connection to the database is granted to the user.

Each of the three statements in next example respectively performs one of the following operations on a role:

- Declares a role called **Engineer**
- Assigns Select privileges on the **locomotives** table to the **Engineer** role
- Defines **Engineer** as the default role for the user **jgould**.

```
EXEC SQL CREATE ROLE 'Engineer';
EXEC SQL GRANT SELECT ON locomotives TO 'Engineer';
EXEC SQL GRANT DEFAULT ROLE 'Engineer' TO jgould;
```

If **jgould** subsequently uses the SET ROLE statement to enable some other role, then by executing the following statement, **jgould** replaces that role with **Engineer** as the current role:

```
SET ROLE DEFAULT;
```

If you have no default role, SET ROLE DEFAULT makes NONE your current role, leaving only the privileges that have been granted explicitly to your *username* or to PUBLIC. After GRANT DEFAULT ROLE changes your default role to a new default role, executing SET ROLE DEFAULT restores your most recently granted default role, even if this role was not your default role when you connected to the database.

If one default role is granted to PUBLIC, but a different role is granted as the default role to an individual user, the individually-granted default role takes precedence if that user issues SET ROLE DEFAULT or connects to the database.

SET SESSION AUTHORIZATION statement

The SET SESSION AUTHORIZATION statement lets you change the user name under which database operations are performed in the current session.

Syntax

SET SESSION AUTHORIZATION TO { *'user_identifier'* | *user_ID_variable* } [**USING** { *password* | *auth_variable* }]

Element	Description	Restrictions	Syntax
auth_variable	Host variable that holds the valid password for the login name specified in *user_identifier* or *user_ID_variable.*	Variable must be a fixed-length character data type. Its value has the same restrictions as *password*.	Must conform to language-specific rules for variable names.
password	Quoted string that is the password of the specified user.	Must be the password of that user, and no more than 32 bytes	Quoted String on page 1202
user_identifier	Quoted string that is a valid login name for the application. The quotation mark delimiters preserve the lettercase.	Authorization identifier of no more than 32 bytes	Quoted String on page 1202
user_ID_variable	The name of an ESQL/C host variable that holds the value of a user identifier.	Variable must be a fixed-length character	Must conform to language-specific

Element	Description	Restrictions	Syntax
		data type. Its value has the same restrictions as *user_identifier*.	rules for variable names.

Usage

This statement allows you to assume the identity of another user, including the discretionary access control (DAC) and label-based access control (LBAC) credentials. You can also use this statement in an API that supports Informix® trusted contexts to switch the user ID on a trusted connection.

Both the DBA and SETSESSIONAUTH access privilege are required to execute this statement. Unless when you start the session you already hold the SETSESSIONAUTH privilege for PUBLIC (or for the user whose name you specify in the SET SESSION AUTHORIZATION statement), and you also hold the DBA privilege, this statement fails with an error.

If the database server has been converted from a legacy version that did not support label-based access control, users who held the DBA privilege are automatically granted the SETSESSIONAUTH access privilege for PUBLIC in the migration process. If the database server has been initialized as a version that supports LBAC security policies, users who hold the **DBSECADM** role can grant the SETSESSIONAUTH privilege to other users. Because the security credentials of each user determine what data rows can be accessed in protected tables, the **DBSECADM** should exercise care in granting the SETSESSIONAUTH privilege and in specifying its scope.

The new identity remains in effect in the current database until you execute SET SESSION AUTHORIZATION again, or until you close the current database. When you use this statement, the specified *user* must have the Connect privilege on the current database. In addition, the DBA cannot set the new authorization identifier to the PUBLIC group, nor to any existing role in the current database.

Setting a session to another user causes a change in a user name in the current active database server. The specified *user*, as far as this database server process is concerned, is completely dispossessed of any privileges while accessing the database server through some administrative utility. Additionally, the new session *user* is not able to initiate any administrative operation (execute a utility, for example) by virtue of the acquired identity.

After the SET SESSION AUTHORIZATION statement successfully executes, any role enabled by a previous user is relinquished. You must use the SET ROLE statement if you wish to assume a role that has been granted to the specified *user*. The database server does not enable the default role of *user* automatically.

After SET SESSION AUTHORIZATION successfully executes, the database server puts any owner-privileged UDRs that the DBA created while using the new authorization identifier in RESTRICTED mode, which can affect access privileges during operations of the UDR on objects in remote databases. For more information on RESTRICTED mode, see the **sysprocedures** system catalog table in the *HCL® Informix® Guide to SQL: Reference*.

When you assume the identity of another user by executing the SET SESSION AUTHORIZATION statement, you can perform operations in the current database only. You cannot perform an operation on a database object outside the current database, such as a remote table. In addition, you cannot execute a DROP DATABASE or RENAME DATABASE statement, even if the database is owned by the real user or by the effective user.

You can use this statement either to obtain access to the data directly or to grant the database-level or table-level privileges needed for the database operation to proceed. The following example shows how to use the SET SESSION AUTHORIZATION statement to obtain table-level privileges:

```
SET SESSION AUTHORIZATION TO 'cathl';
GRANT ALL ON customer TO 'mary';
SET SESSION AUTHORIZATION TO 'mary';
UPDATE customer SET fname = 'Carl' WHERE lname = 'Pauli';
```

If you enclose *user* in quotation marks, the name is case sensitive and is stored exactly as you typed it. In an ANSI-compliant database, if you do not use quotation marks as delimiters, the authorization identifier is stored in uppercase letters, unless the **ANSIOWNER** environment variable is set to prevent the conversion of lowercase letters to uppercase.

The following Open Database Connectivity (ODBC) API example enables user ID switching on a trusted connection with an authentication requirement:

```
SQLExecDirect(hstmt,"SET SESSION AUTHORIZATION TO 'zurbie' USING 'pass01'",SQL_NTS);
```

In the function call above,

- `'zurbie'` specifies the authorization identifier for subsequent operations in this session
- `pass01` must be the current password of user **zurbie**.

Note:

Except in a non-hostile environment, `'pass01'` is not a recommended example of a login password, because in some locales it might be easy to guess.

SET SESSION AUTHORIZATION and Transactions

If your database is not ANSI compliant, you must issue the SET SESSION AUTHORIZATION statement outside a transaction. If you issue the statement within a transaction, you receive an error message.

In an ANSI-compliant database, you can execute the SET SESSION AUTHORIZATION statement only if you have not executed a statement that initiates an implicit transaction (for example, CREATE TABLE or SELECT). Statements that do not initiate an implicit transaction are statements that do not acquire locks or log data (for example, SET EXPLAIN and SET ISOLATION). You can execute the SET SESSION AUTHORIZATION statement immediately after a DATABASE statement or a COMMIT WORK statement.

SET STATEMENT CACHE statement

Use the SET STATEMENT CACHE statement to turn on caching or turn off caching for the current session. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
SET STATEMENT CACHE { ON  | OFF }
```

Usage

You can use the SET STATEMENT CACHE statement to turn caching in the SQL statement cache ON or OFF for the current session. The statement cache stores in a buffer identical statements that are repeatedly run in a session. Only data manipulation language (DML) statements (DELETE, INSERT, UPDATE, or SELECT) can be stored in the statement cache.

This mechanism allows qualifying statements to bypass the optimization stage, and avoid recompiling, which can reduce memory consumption and can improve query processing time.

Examples

The following example turns on statement caching for the current session:

```
SET STATEMENT CACHE ON;
```

The following turns off statement caching for the current session:

```
SET STATEMENT CACHE OFF;
```

Precedence and Default Behavior

SET STATEMENT CACHE takes precedence over the **STMT_CACHE** environment variable and the STMT_CACHE configuration parameter. You must enable the SQL statement cache, however, either by setting the STMT_CACHE configuration parameter or by using the **onmode** utility, before the SET STATEMENT CACHE statement can execute successfully.

When you issue a SET STATEMENT CACHE ON statement, the SQL statement cache remains in effect until you issue a SET STATEMENT CACHE OFF statement or until the program ends. If you do not use SET STATEMENT CACHE, the default behavior depends on the setting of the **STMT_CACHE** environment variable or the STMT_CACHE configuration parameter.

Turning the Cache ON

Use the ON option to enable the SQL statement cache. When the SQL statement cache is enabled, each statement that you execute passes through the SQL statement cache to determine if a matching cache entry is present. If so, the database server uses the cached entry to execute the statement.

If the statement has no matching entry, the database server tests to see if it qualifies for entry into the cache. For the conditions a statement must meet to enter into the cache, see SQL statement cache qualifying criteria on page 835.

Restrictions on Matching Entries in the SQL Statement Cache

When the database server considers whether or not a statement is identical to a statement in the SQL statement cache, the following items must match:

- Lettercase
- Comments
- White space
- Optimization settings

    - SET OPTIMIZATION statement options
    - Optimizer directives
    - The SET ENVIRONMENT OPTCOMPIND statement options or settings of the **OPTCOMPIND** environment variable, or of the OPTCOMPIND configuration parameter in the ONCONFIG file. (If conflicting settings exist for the same query, this is the descending order of precedence.)
- Parallelism settings
    - SET PDQPRIORITY statement options or settings of the **PDQPRIORITY** environment variable
- Query text strings
- Literals

If an SQL statement is semantically equivalent to a statement in the SQL statement cache but has different literals, the statement is not considered identical and qualifies for entry into the cache. For example, the following SELECT statements are not identical:

```
SELECT col1, col2 FROM tab1 WHERE col1=3;

SELECT col1, col2 FROM tab1 WHERE col1=5;
```

In this example, both statements are entered into the SQL statement cache.

Host-variable names, however, are insignificant. For example, the following select statements are considered identical:

```
SELECT * FROM tab1 WHERE x = :x AND y = :y;

SELECT * FROM tab1 WHERE x = :p AND y = :q;
```

In the previous example, although the host names are different, the statements qualify, because the case, query text strings, and white space match. Performance does not improve, however, because each statement has already been parsed and optimized by the PREPARE statement.

Turning the Cache OFF

The OFF option disables the SQL statement cache. When you turn caching OFF for your session, no SQL statement cache code is executed for that session.

The SQL statement cache is designed to save memory in environments where identical queries are executed repeatedly and schema changes are infrequent. If this is not the case, you might want to turn the SQL statement cache off to avoid the overhead of caching. For example, if you have little cache cohesion, that is, when relatively few matches but many new entries into the cache exist, the cache management overhead is high. In this case, turn the SQL statement cache off.

If you know that you are executing many statements that do not qualify for the SQL statement cache, you might want to disable it and avoid the overhead of testing to see if each DML statement qualifies for insertion into the cache.

SQL statement cache qualifying criteria

A statement that can be cached in the SQL statement cache (and consequently, one that can match a statement that already appears in the SQL statement cache) must meet specified conditions.

To quality for caching, the statement must meet all of the following conditions:

- It must be a SELECT, INSERT, UPDATE, or DELETE statement.
- It must contain only non-opaque built-in data types (excluding BLOB, BOOLEAN, BYTE, CLOB, LVARCHAR, and TEXT).
- It must contain only built-in operators.
- It cannot contain user-defined routines.
- It cannot contain temporary or remote tables.
- It cannot contain subqueries in the Projection list.
- It cannot be part of a multistatement PREPARE.
- It cannot have user-privilege restrictions on target columns.
- In an ANSI-compliant database, it must contain fully qualified object names.
- It cannot require re-optimization.

Requiring Re-Execution Before Cache Insertion

A qualified SQL statement is fully inserted into the SQL statement cache only after the database server counts a configurable number of references (which are sometimes called "hits") to that statement. For the default value of zero, a qualified DML statement does not need to be re-executed before it is cached.

Using the STMT_CACHE_HITS configuration parameter, however, the database administrator (DBA) can specify that qualified DML statements must be executed a minimum number of times before they are inserted into the statement cache. By setting this to a value of one (or to a larger value), the DBA excludes one-time-only *ad hoc* queries from full insertion into the SQL statement cache, thereby lowering cache-management overhead.

Enabling or Disabling Insertions After Size Exceeds Configured Limit

The DBA can prevent the insertion of additional qualified SQL statements into the statement cache when the cache size reaches its configured size (as specified by the STMT_CACHE_SIZE configuration parameter) by setting the configuration parameter STMT_CACHE_NOLIMIT to zero.

Prepared Statements and the Statement Cache

Prepared statements are inherently cached for a single session. That is, if a prepared statement is executed many times (or if a single cursor is opened many times), the same prepared query plan is used repeatedly by that session.

If a session prepares a statement and then executes it many times, its performance is essentially unaffected by using the SQL statement cache, because the statement is optimized just once, during the PREPARE statement.

If other sessions also prepare that same statement, however, or if the first session prepares the statement several times, then the statement cache usually provides a direct performance benefit, because the database server only calculates the query plan once. Of course, the original session might gain a small benefit from the statement cache, even if it prepares the statement only once, because other sessions use less memory, and the database server does less work for the other sessions.

SET TRANSACTION statement

Use the SET TRANSACTION statement to define the isolation level and to specify whether the access mode of a transaction is read-only or read-write.

Syntax

```
(explicit id unique_1173_Connect_42_settr002) unique_1173_Connect_42_settr002 SET TRANSACTION { | [ { READ WRITE | READ ONLY } | [ ISOLATION LEVEL { READ COMMITTED | REPEATABLE READ | SERIALIZABLE | READ UNCOMMITTED } }
```

Usage

SET TRANSACTION is valid only in databases with transaction logging. You can issue this statement from a client computer only after a database is opened. The transaction isolation level affects concurrency among processes that attempt to access the same rows simultaneously from the database. The database server uses shared locks to support different levels of isolation among processes that are attempting to read data, as the following list shows:

- Read Uncommitted
- Read Committed
- (ANSI) Repeatable Read
- Serializable

The update or delete process always acquires an exclusive lock on the row that is being modified. The level of isolation does not interfere with such rows, but the access mode does affect whether you can update or delete rows.

If another process attempts to update or delete rows that you are reading with an isolation level of Serializable or (ANSI) Repeatable Read, that process will be denied access to those rows.

Comparing SET TRANSACTION with SET ISOLATION

The SET TRANSACTION statement complies with ANSI SQL-92. This statement is similar to the Informix® SET ISOLATION statement; however, the SET ISOLATION statement is not ANSI compliant and does not provide access modes. In fact, the isolation levels that you can set with the SET TRANSACTION statement are almost parallel to the isolation levels that you can set with the SET ISOLATION statement, as the following table shows.

SET TRANSACTION Isolation Level	SET ISOLATION Isolation Level
Read Uncommitted	Dirty Read
Read Committed	Committed Read
[*Not supported*]	Cursor Stability
(ANSI) Repeatable Read	(Informix®) Repeatable Read
Serializable	(Informix®) Repeatable Read

Another difference between SET TRANSACTION and SET ISOLATION is the behavior of the isolation levels within transactions. You can issue SET TRANSACTION only once for a transaction. Any cursors that are opened during that

transaction are guaranteed that isolation level (or access mode, if you are defining an access mode). With SET ISOLATION, after a transaction is started, you can change the isolation level more than once within the transaction.

The following examples illustrate this difference in the behavior of the SET ISOLATION and SET TRANSACTION statements:

```
EXEC SQL BEGIN WORK;
EXEC SQL SET ISOLATION TO DIRTY READ;
EXEC SQL SELECT ... ;
EXEC SQL SET ISOLATION TO REPEATABLE READ;
EXEC SQL INSERT ... ;
EXEC SQL COMMIT WORK;    -- Executes without error
```

Compare the previous example to these SET TRANSACTION statements:

```
EXEC SQL BEGIN WORK;
EXEC SQL SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
EXEC SQL SELECT ... ;
EXEC SQL SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
   -- Produces error 876: Cannot issue SET TRANSACTION
   -- in an active transaction.
```

An additional difference between SET ISOLATION and SET TRANSACTION is the duration of isolation levels. Because SET ISOLATION supports complete-connection level settings, the isolation level specified by SET ISOLATION remains in effect until another SET ISOLATION statement is issued. The isolation level set by SET TRANSACTION only remains in effect until the transaction terminates. Then the isolation level is reset to the default for the database type.

Informix® Isolation Levels

The following definitions explain the critical characteristics of each isolation level, from the lowest level of isolation to the highest.

Using the Read Uncommitted Option

Use the Read Uncommitted option to copy rows from the database whether or not locks are present on them. The program that fetches a row places no locks and it respects none. Read Uncommitted is the only isolation level available to databases that do not have transactions.

This isolation level is most appropriate in queries of static tables whose data is not being modified, because it provides no isolation. With Read Uncommitted, the program might return an uncommitted row that was inserted or modified within a transaction that was subsequently rolled back.

The Uncommitted Read isolation level of SET TRANSACTION does not directly support the LAST COMMITTED feature of the Committed Read isolation level of the SET ISOLATION statement. The LAST COMMITTED feature can reduce the risk of locking conflicts when an application attempts to read a row on which another session holds an exclusive lock while modifying data. When this feature is enabled, the database server returns the most recently committed version of the data, rather than wait for the lock to be released.

This feature takes effect implicitly, however, in all user sessions that use the Read Uncommitted isolation level of the SET TRANSACTION statement, under either of the following circumstances:

- if the USELASTCOMMITTED configuration parameter is set to `'DIRTY READ'` or to `'ALL'`
- if the SET ENVIRONMENT statement set the USELASTCOMMITTED session environment option to `'DIRTY READ'` or to `'ALL'`.

See the section #unique_858 for more information about the LAST COMMITTED feature and its restrictions.

Using the Read Committed Option

Use the Read Committed option to guarantee that every retrieved row is committed in the table at the time that the row is retrieved. This option does not place a lock on the fetched row. Read Committed is the default level of isolation in a database with logging that is not ANSI compliant.

Read Committed is appropriate when each row of data is processed as an independent unit, without reference to other rows in the same or other tables.

The Read Committed isolation level of SET TRANSACTION does not directly support the LAST COMMITTED feature of the Committed Read isolation level of the SET ISOLATION statement, which can reduce the risk of locking conflicts when an application attempts to read data in a row on which another session holds an exclusive row-level lock. When this feature is enabled, the database server returns the most recently committed version of the data, rather than wait for the lock to be released

This feature takes effect implicitly, however, in all user sessions that use the Read Committed isolation level of the SET TRANSACTION statement, under either of the following circumstances:

- if the USELASTCOMMITTED configuration parameter is set to `'COMMITTED READ'` or to `'ALL'`
- if the SET ENVIRONMENT statement set the USELASTCOMMITTED session environment variable to `'COMMITTED READ'` or to `'ALL'`.

See the section #unique_858 for more information about the LAST COMMITTED feature and its restrictions.

Using the Repeatable Read and Serializable Options

The Informix® implementation of Repeatable Read and of Serializable are equivalent. The Serializable (or Repeatable Read) option places a shared lock on every row that is selected during the transaction.

Another process can also place a shared lock on a selected row, but no other process can modify any selected row during your transaction or insert a row that meets the search criteria of your query during your transaction.

A *phantom row* is a row that was not visible when you first read the query set, but that materializes in a subsequent read of the query set in the same transaction. Only this isolation level prevents access to a phantom row.

If you repeat the query during the transaction, you reread the same data. The shared locks are released only when the transaction is committed or rolled back. Serializable is the default isolation level in an ANSI-compliant database. Serializable isolation places the largest number of locks and holds them the longest. Therefore, it is the level that reduces concurrency the most.

Default Isolation Levels

The default isolation level is established when you create the database.

Informix® Name	ANSI Name	When This Is the Default Level of Isolation
Dirty Read	Read Uncommitted	Database without transaction logging
Committed Read	Read Committed	Databases with logging that are not ANSI- compliant
Repeatable Read	Serializable	ANSI-compliant databases

For Informix® databases that are not ANSI-compliant, unless you explicitly set the USELASTCOMMITTED configuration parameter, the LAST COMMITTED feature is not in effect for the default isolation levels. The SET ENVIRONMENT statement or the SET ISOLATION statement can override this default and enable LAST COMMITTED for the current session.

The default isolation level remains in effect until you issue a SET TRANSACTION statement within a transaction. After a COMMIT WORK statement completes the transaction or a ROLLBACK WORK statement cancels the entire transaction, the isolation level is reset to the default.

When you use High Availability Data Replication, the database server effectively uses Dirty Read isolation on the HDR Secondary Server, regardless of the specified **SET ISOLATION** or **SET TRANSACTION** isolation level, unless the UPDATABLE_SECONDARY configuration parameter is enabled. For more information about this topic, see Isolation Levels for Secondary Data Replication Servers on page 817.

Access Modes

Access modes affect read and write concurrency for rows within transactions. Use access modes to control data modification. SET TRANSACTION can specify that a transaction is read-only or read-write. By default, transactions are read-write. When you specify a read-only transaction, certain limitations apply. Read-only transactions cannot perform the following actions:

- Insert, delete, or update rows of a table.
- Create, alter, or drop any database object such as schemas, tables, temporary tables, indexes, or SPL routines.
- Grant or revoke access privileges.
- Update statistics.
- Rename columns or tables.

You can execute SPL routines in a read-only transaction as long as the SPL routine does not try to perform any restricted statement.

Effects of Isolation Levels

You cannot set the transaction isolation level in a database that does not have logging. Every retrieval in an unlogged database occurs as a Read Uncommitted.

The data that is obtained during retrieval of BYTE or TEXT data can vary, depending on the transaction isolation levels. Under Read Uncommitted or Read Committed isolation levels, a process is permitted to read a BYTE or TEXT column that is either deleted (if the delete is not yet committed) or in the process of being deleted. Under these isolation levels, an application can read a deleted BYTE or TEXT column when certain conditions exist. For information about these conditions, see the *HCL® Informix® Administrator's Guide*.

In , if you use a scroll cursor in a transaction, you can force consistency between your temporary table and the database table either by setting the isolation level to Serializable or by locking the entire table. A scroll cursor with hold, however, cannot guarantee the same consistency between the two tables. Table-level locks set by Serializable are released when the transaction is completed, but the scroll cursor with hold remains open beyond the end of the transaction. You can modify released rows as soon as the transaction ends, so the retrieved data in the temporary table might be inconsistent with the actual data.

Warning: Do not use nonlogging tables within a transaction. If you need to use a nonlogging table within a transaction, either set the isolation level to Repeatable Read or lock the table in exclusive mode to prevent concurrency problems.

SET Transaction Mode statement

Use the SET Transaction Mode statement to specify whether constraints are checked at the statement level or at the transaction level during the current transaction.

Syntax

```
SET CONSTRAINTS { constraint | ALL } { IMMEDIATE | DEFERRED }
```

Element	Description	Restrictions	Syntax
constraint	Constraint whose transaction mode is to be changed	All constraints must exist in the same database, which must support logging	Identifier on page 1231

Usage

To enable or disable constraints, or to change their filtering mode, see SET Database Object Mode statement on page 784.

This statement is valid only in a database with transaction logging, and its effect is limited to the transaction in which it is executed.

Use the IMMEDIATE keyword to set the transaction mode of constraints to statement-level checking. IMMEDIATE is the default transaction mode of constraints when they are created.

Use the DEFERRED keyword to set the transaction mode to transaction-level checking. You cannot change the transaction mode of a constraint to DEFERRED unless the constraint is currently enabled.

Statement-Level Checking

When you set the transaction mode to IMMEDIATE, statement-level checking is turned on, and all specified constraints are checked at the end of each INSERT, UPDATE, or DELETE statement. If a constraint violation occurs, the statement is not executed.

Transaction-Level Checking

When you set the transaction mode of constraints to DEFERRED, statement-level checking is turned off, and all (or the specified) constraints are not checked until the transaction is committed. If a constraint violation occurs while the transaction is being committed, the transaction is rolled back.



Tip: If you defer checking a primary-key constraint, checking the not-NULL constraint for that column or set of columns is also deferred.

Duration of Transaction Modes

The duration of the transaction mode that the SET Transaction Mode statement specifies is the transaction in which the SET Transaction Mode statement is executed. You cannot execute this statement outside a transaction. Once a COMMIT WORK or ROLLBACK WORK statement is successfully completed, the transaction mode of all constraints reverts to IMMEDIATE.

To switch from transaction-level checking to statement-level checking, you can use the SET Transaction Mode statement to set the transaction mode to IMMEDIATE, or you can use a COMMIT WORK or ROLLBACK WORK statement to terminate your transaction.

Specifying All Constraints or a List of Constraints

You can specify all constraints in the database in the SET Transaction Mode statement, or you can specify a single constraint, or list of constraints.

If you specify the ALL keyword, the SET Transaction Mode statement sets the transaction mode for all constraints in the database. If any statement in the transaction requires that any constraint on any table in the database be checked, the database server performs the checks at the statement level or the transaction level, depending on the setting that you specify in the SET Transaction Mode statement.

If you specify a single constraint name or a list of constraints, the SET Transaction Mode statement sets the transaction mode for the specified constraints only. If any statement in the transaction requires checking of a constraint that you did not specify in the SET Transaction Mode statement, that constraint is checked at the statement level regardless of the setting that you specified in the SET Transaction Mode statement for other constraints.

When you specify a list of constraints, the constraints do not need to be defined on the same table, but they must exist in the same database.

Specifying Remote Constraints

You can set the transaction mode of local constraints or remote constraints. That is, the constraints that are specified in the SET Transaction Mode statement can be constraints that are defined on local tables or constraints that are defined on remote tables.

Examples of Setting the Transaction Mode for Constraints

The following example shows how to defer checking constraints within a transaction until the transaction is complete. The SET Transaction Mode statement in the example specifies that any constraints on any tables in the database are not checked until the COMMIT WORK statement is encountered.

```
BEGIN WORK;
SET CONSTRAINTS ALL DEFERRED;
...
COMMIT WORK;
```

The following example specifies that a list of constraints is not checked until the transaction is complete:

```
BEGIN WORK;
SET CONSTRAINTS update_const, insert_const DEFERRED;
...
COMMIT WORK;
```

SET TRIGGERS statement

Use the SET TRIGGERS statement to enable or disable all or some of the triggers on a table, or all or some of the INSTEAD OF triggers on a view.

Syntax

```
SET TRIGGERS { trigger | FOR [ 'owner' . ] { table | view } } { ENABLED | [ { DISABLED } ] }
```

Element	Description	Restrictions	Syntax
owner	The owner of *table* or *view*	Must own the table or view	Owner name on page 1262
table	Table whose triggers are all to be enabled or disabled	Must exist	Identifier on page 1231
trigger	Trigger to be enabled or disabled	Must exist	Identifier on page 1231
view	View whose INSTEAD OF triggers are all to be enabled or disabled	Must exist	Identifier on page 1231

Usage

The SET TRIGGERS statement is a special case of the SET Database Object Mode statement. The SET Database Object Mode statement can also enable or disable an index or a constraint, or change the filtering mode of a unique index or of a constraint.

For the complete syntax and semantics of the SET TRIGGERS statement, see SET Database Object Mode statement on page 784.

Restrictions on secondary servers

In cluster environments, the SET TRIGGERS statement is not supported on updatable secondary servers. (More generally, session-level index, trigger, and constraint modes that the SET Database Object Mode statement specifies are not redirected for UPDATE operations on table objects in databases of secondary servers.)

SET USER PASSWORD statement (UNIX™, Linux™)

Use the SET USER PASSWORD statement to change your password for database server access if you are an internally authenticated user. This statement is an extension to the ANSI/ISO standard for the SQL language.

Syntax

```
SET USER PASSWORD OLD old_password NEW new_password
```

Element	Description	Restrictions	Syntax
new_password	New password for internal authentication of the user.	Length must be between 6 and 32 bytes.	Quoted String on page 1202
old_password	Existing password for internal authentication of the user.	Length must be between 6 and 32 bytes.	Quoted String on page 1202

Usage

A DBSA cannot use this statement to change the password of another user. To change the passwords of other users, a DBSA can use the ALTER USER statement.

Execution of the SET USER PASSWORD statement can be audited with the PWUR audit code.

Example

Example

The following statement changes the password from **joebar** to **joefoo**:

```
SET USER PASSWORD OLD 'joebar' NEW 'joefoo';
```

START VIOLATIONS TABLE statement

Use the START VIOLATIONS TABLE statement to create a violations table and a diagnostics table for a specified target table. The START VIOLATIONS TABLE statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
START VIOLATIONS TABLE FOR [ 'owner' . ] table [ USING violations, diagnostics ] [ MAX ROWS num_rows ]
```

Element	Description	Restrictions	Syntax
diagnostics	Declares the name of a diagnostics table to be associated with the target *table*. Default name is **table_dia**.	Must be unique among names of tables, views, sequences, and synonyms	Identifier on page 1231
num_rows	Maximum number of rows that the database server can insert into *violations* when a single statement is executed on *table*	Must be an integer in range from 1 to the maximum value of the INTEGER data type	Literal Number on page 1198
owner	The owner of *table*	Must own the table	Owner name on page 1262
table	Target table for which *violations* and *diagnostics* tables are to be created	If USING clause is omitted, no more than 124 bytes	Identifier on page 1231
violations	Violations table to be associated with *table*. Default name is **table_vio**.	Same restrictions as *diagnostics*	Identifier on page 1231

Usage

The database server associates the *violations* table and the *diagnostics* table) with the target table that you specify after the FOR keyword by recording the relationship among the three tables in the **sysviolations** system catalog table.

A target table must satisfy these requirements:

- It cannot be a table in a database that is not the current database.
- It cannot be an object that the CREATE EXTERNAL TABLE statement defined.
- It cannot already be associated with a violations or diagnostics table.
- It cannot be a system catalog table.

The START VIOLATIONS TABLE statement creates the special violations table that holds nonconforming rows that fail to satisfy constraints and unique indexes during insert, update, and delete operations on target tables. This statement also creates the special diagnostics table that contains information about the integrity violations that each row causes in the violations table.

Relationship to the SET Database Object Mode statement

The START VIOLATIONS TABLE statement is closely related to the SET Database Object Mode statement. If you use SET Database Object Mode to set the constraints or unique indexes defined on a table to the FILTERING mode, without also using START VIOLATIONS TABLE, any rows that violate a constraint or unique-index requirement in data manipulation operations are not filtered out to a violations table. Instead you receive an error message that indicates that you must start a violations table for the target table.

Similarly, if you use the SET Database Object Mode statement to set a disabled constraint or disabled unique index to the ENABLED or FILTERING mode, but you do not use START VIOLATIONS TABLE for the table on which the database objects are defined, any rows that do not satisfy the constraint or unique-index requirement are not filtered out to a violations table.

In these cases, to identify the rows that do not satisfy the constraint or unique-index requirement, issue the START VIOLATIONS TABLE statement to start the violations and diagnostics tables. Do this before you use the SET Database Object Mode statement to set the database objects to the ENABLED or FILTERING database object mode.

Effect on concurrent transactions

If the database has transaction logging, you must issue START VIOLATIONS TABLE in isolation. That is, no other transaction can be in progress on a target table when you issue START VIOLATIONS TABLE on that table within a transaction. Any transactions that start on the target table after the first transaction has issued the START VIOLATIONS TABLE statement will behave the same way as the first transaction with respect to the violations and diagnostics tables. That is, any constraint and unique-index violations by these subsequent transactions will be recorded in the violations and diagnostics tables.

For example, if transaction A operates on table **tab1** and issues a START VIOLATIONS TABLE statement on table **tab1**, the database server starts a violations table named **tab1_vio** and filters any constraint or unique-index violations on table **tab1** by transaction A to table **tab1_vio**. If transactions B and C start on table **tab1** after transaction A has issued the START VIOLATIONS TABLE statement, the database server also filters any constraint and unique-index violations by transactions B and C to table **tab1_vio**.

The result is that all three transactions do not receive error messages about constraint and unique-index violations, even though transactions B and C do not expect this behavior. For example, if transaction B issues an INSERT or UPDATE statement that violates a check constraint on table **tab1**, the database server does not issue a constraint violation error to transaction B. Instead, the database server filters the nonconforming row (also called a "bad row") to the violations table without notifying transaction B that a data-integrity violation occurred.

You can prevent this situation from arising in Informix® by specifying WITH ERRORS when you specify the FILTERING mode in a SET Database Object Mode, CREATE TABLE, ALTER TABLE, or CREATE INDEX statement. When multiple transactions operate on a table and the WITH ERRORS option is in effect, any transaction that violates a constraint or unique-index requirement on a target table receives a data-integrity error message.

Stopping the Violations and Diagnostics Tables

After you use START VIOLATIONS TABLE to create an association between a target table and the violations and diagnostics tables, the only way to drop the association between the target table and the violations and diagnostics tables is to issue a STOP VIOLATIONS TABLE statement for the target table. For more information, see STOP VIOLATIONS TABLE statement on page 856.

USING Clause

You can use the USING clause to declare explicit names for the violations table and for the diagnostics table.

If you omit the USING clause, the database server assigns names to the violation table and the diagnostics table. The system-assigned name of the violations table consists of the name of the target table followed by the string `_vio`. The name that the database server assigns to the diagnostics table consists of the name of the target table followed by the string `_dia`.

If you omit the USING clause, the maximum length of the name of the target table is 124 bytes.

Using the MAX ROWS clause

The MAX ROWS clause specifies the maximum number of rows that the database server can insert into the diagnostics table when a single statement is executed on the target table. If you omit the MAX ROWS clause, no upper limit is imposed on the number of rows that can be inserted into the diagnostics table when a single statement is executed on the target table.

Specifying the maximum number of rows in the diagnostics table

The following statement starts violations and diagnostics tables for the target table named **orders**. The MAX ROWS clause specifies the maximum number of rows that can be inserted into the diagnostics table when a single statement, such as an INSERT statement, is executed on the target table.

```
START VIOLATIONS TABLE FOR orders MAX ROWS 50000;
```

Privileges required for starting violations or diagnostics tables

To start a violations or a diagnostics table for a target table, you must meet one of the following requirements:

- You must have the DBA privilege on the database.
- You must be the owner of the target table and also have the Resource privilege on the database.
- You must have the Alter privilege on the target table and also have the Resource privilege on the database.

Structure of the violations table

When you issue START VIOLATIONS TABLE for a target table, the violations table that the statement creates has a predefined structure. This structure consists of the columns of the target table and three additional columns.

The following table shows the schema of the violations table.

Column Name	Data Type	Column Description
Same columns (in the same order) that appear in the target table	Same types as corresponding columns in the target table.	The violations table has the same schema as the target table, so that rows violating constraints or a unique-index during insert, update, and delete operations can be filtered to the violations table.
informix_tupleid	SERIAL	Unique serial code for the nonconforming row
informix_optype	CHAR(1)	The type of operation that caused this bad row. This column can have the following values: I = Insert D = Delete O = Update (with original values in this row) N = Update (with new values in this row) S = SET Database Object Mode

Column Name	Data Type	Column Description
informix_recowner	CHAR(32)	User who issued the statement that created this nonconforming row

If the target table of the START VIOLATIONS TABLE statement is protected by a security policy, the database server protects the violations table with same security policy. In this case, the schema of the violations table includes an IDSSECURITYLABEL column whose name and position among other columns corresponds to the IDSSECURITYLABEL column of the target table. When the violations table is created, any SECURED WITH *label* specifications that protect columns in the target table also protect the corresponding violations table columns.

Serial columns in the target table are converted to integer data types in the violations table.

Users can examine these nonconforming rows in the violations table, analyze the related rows that contain diagnostic information in the diagnostics table, and take corrective actions.

Examples of START VIOLATIONS TABLE Statements

The following examples show different ways to execute the START VIOLATIONS TABLE statement.

Violations and Diagnostics Tables with Default Names

The following statement starts violations and diagnostics tables for the target table named **cust_subset**. The violations table is named **cust_subset_vio** by default, and the diagnostics table is named **cust_subset_dia** by default.

```
START VIOLATIONS TABLE FOR cust_subset;
```

Violations and Diagnostics Tables with Explicit Names

The following statement starts a violations and diagnostics table for the target table named **items**. The USING clause assigns explicit names to the violations and diagnostics tables. The violations table is to be named **exceptions**, and the diagnostics table is to be named **reasons**.

```
START VIOLATIONS TABLE FOR items USING exceptions, reasons;
```

Relationships Among the Target, Violations, and Diagnostics Tables

Users can take advantage of the relationships among the target, violations, and diagnostics tables to obtain diagnostic information about rows that cause data-integrity violations during INSERT, DELETE, and UPDATE statements. Each row of the violations table has at least one corresponding row in the diagnostics table.

- One row in the violations table is a copy of any row in the target table for which a data-integrity violation was detected. A row in the diagnostics table contains information about the nature of the data-integrity violation caused by the nonconforming row in the violations table.
- One row in the violations table has a unique serial identifier in the **informix_tupleid** column. A row in the diagnostics table has the same serial identifier in its **informix_tupleid** column.

A given row in the violations table can have more than one corresponding row in the diagnostics table. The multiple rows in the diagnostics table all have the same serial identifier in their **informix_tupleid** column so that they are all linked to the same row in the violations table. Multiple rows can exist in the diagnostics table for the same row in the violations table because a nonconforming row in the violations table can cause more than one data-integrity violation.

For example, the same nonconforming row can violate a unique index for one column, a not-NULL constraint for another column, and a check constraint for a third column. In this case, the diagnostics table contains three rows for the single nonconforming row in the violations table. Each of these diagnostic rows identifies a different data-integrity violation that the nonconforming row in the violations table caused.

By joining the violations and diagnostics tables, the DBA or target-table owner can obtain diagnostic information about any or all nonconforming rows in the violations table. SELECT statements can perform these joins interactively, or you can write a program to perform them within transactions.

Initial Privileges on the Violations Table

When you issue the START VIOLATIONS TABLE statement to create the violations table, the database server uses the set of privileges granted on the target table as a basis for granting privileges on the violations table. The database server follows different rules, however, when it grants each type of privilege.

The following table summarizes the circumstances under which the database server grants each type of privilege on the violations table.

Privilege

Condition for Granting the Privilege

Alter

Alter privilege is not granted on the violations table. (Users cannot alter violations tables.)

Index

User has Index privilege on the violations table if the user has the Index privilege on the target table.

Insert

User has the Insert privilege on the violations table if the user has the Insert, Delete, or Update privilege on any column of the target table.

Delete

User has the Delete privilege on the violations table if the user has the Insert, Delete, or Update privilege on any column of the target table.

Select

User has the Select privilege on the **informix_tupleid**, **informix_optype**, and **informix_recowner** columns of the violations table if the user has the Select privilege on any column of the target table.

User has the Select privilege on any other column of the violations table if the user has the Select privilege on the same column in the target table.

Update

User has the Update privilege on the **informix_tupleid**, **informix_optype**, and **informix_recowner** columns of the violations table if the user has the Update privilege on any column of the target table.

(Even with the Update privilege on the **informix_tupleid** column, however, the user cannot update this SERIAL column.)

User has the Update privilege on any other violations table column if the user has the Update privilege on the same column in the target table.

References

The References privilege is not granted on the violations table. (Users cannot add referential constraints to violations tables.)

The following rules apply to ownership of the violations table and privileges on the violations table:

- When the violations table is created, the owner of the target table becomes the owner of the violations table.
- The owner of the violations table automatically receives all table-level privileges on the violations table, including the Alter and References privileges. The database server, however, prevents the owner of the violations table from altering the violations table or adding a referential constraint to the violations table.
- You can use the GRANT and REVOKE statements to modify the initial set of privileges on the violations table.
- When you issue an INSERT, DELETE, or UPDATE statement on a target table that has a filtering-mode unique index or constraint defined on it, you must have the Insert privilege on the violations and diagnostics tables.

  If you do not have the Insert privilege on the violations and diagnostics tables, the database server executes the INSERT, DELETE, or UPDATE statement on the target table provided that you have the necessary privileges on the target table. The database server does not return an error concerning the lack of Insert privilege on the violations and diagnostics tables unless an integrity violation is detected during execution of the INSERT, DELETE, or UPDATE statement.

  Similarly, when you issue a SET Database Object Mode statement to set a disabled constraint or disabled unique index to the enabled or filtering mode, and a violations table and diagnostics table exist for the target table, you must have the Insert privilege on the violations and diagnostics tables.

  If you do not have the Insert privilege on the violations and diagnostics tables, the database server executes the SET Database Object Mode statement if you have the necessary privileges on the target table. The database server does not return an error concerning the lack of Insert privilege on the violations and diagnostics tables, unless an integrity violation is detected during the execution of the SET Database Object Mode statement.
- The grantor of the initial set of privileges on the violations table is the same as the grantor of the privileges on the target table.

  For example, if user **henry** was granted the Insert privilege on the target table by both user **jill** and user **albert**, then the Insert privilege on the violations table is granted to **henry** both by **jill** and by **albert**.

- After the violations table is started, revoking a privilege on the target table from a user does not automatically revoke the same privilege on the violations table from that user. Instead, you must explicitly revoke the privilege on the violations table from the user.
- If you have fragment-level privileges on the target table, you have the corresponding fragment-level privileges on the violations table.

Example of Privileges on the Violations Table

The following example illustrates how the initial set of privileges on a violations table is derived from the current set of privileges on the target table. Assume that a table named **cust_subset** consists of the following columns: **ssn** (customer Social Security number), **fname** (customer first name), **lname** (customer last name), and **city** (city in which the customer lives).

The following set of privileges exists on the **cust_subset** table:

- User **barbara** has the Insert and Index privileges on the table. They also have the Select privilege on the **ssn** and **lname** columns.
- User **carrie** has the Update privilege on the **city** column. They also have the Select privilege on the **ssn** column.
- User **danny** has the Alter privilege on the table.

Now user **alvin** starts a violations table named **cust_subset_viols** and a diagnostics table named **cust_subset_diags** for the **cust_subset** table:

```
START VIOLATIONS TABLE FOR cust_subset
   USING cust_subset_viols, cust_subset_diags;
```

The database server grants the following set of initial privileges on the **cust_subset_viols** violations table:

- User **alvin** is the owner of the violations table, so they have all table-level privileges on the table.
- User **barbara** has the Insert, Delete, and Index privileges on the table.

  User **barbara** has the Select privilege on five columns of the violations table: the **ssn**, the **lname**, the **informix_tupleid**, the **informix_optype**, and the **informix_recowner** columns.
- User **carrie** has Insert and Delete privileges on the violations table.

  User **carrie** has the Update privilege on four columns of the violations table: the **city**, the **informix_tupleid**, the **informix_optype**, and the **informix_recowner** columns. They cannot, however, update the **informix_tupleid** column (because this is a SERIAL column).

  User **carrie** has the Select privilege on four columns of the violations table: the **ssn** column, the **informix_tupleid** column, the **informix_optype** column, and the **informix_recowner** column.
- User **danny** has no privileges on the violations table.

Using the Violations Table

The following rules concern the structure and use of the violations table:

- Every pair of update rows in the violations table has the same value in the **informix_tupleid** column to indicate that both rows refer to the same row in the target table.
- If the target table has columns named **informix_tupleid**, **informix_optype**, or **informix_recowner**, the database server attempts to generate alternative names for these columns in the violations table by appending a digit to the end of the column name (for example, **informix_tupleid1**). If this fails, an error is returned, and no violations table is started for the target table.
- When a table functions as a violations table, it cannot have triggers or constraints defined on it.
- When a table functions as a violations table, users can create indexes on it, even though the existence of an index affects performance. Unique indexes on the violations table cannot be set to FILTERING database object mode.
- If a target table has a violations and diagnostics table associated with it, dropping the target table in cascade mode (the default mode) causes the violations and diagnostics tables to be dropped also. If the target table is dropped in the restricted mode, the DROP TABLE operation fails (because the violations and diagnostics tables exist).
- After a violations table is started for a target table, ALTER TABLE cannot add, modify, or drop columns of the violations, diagnostics, or target tables. Before you can alter any of these tables, you must issue a STOP VIOLATIONS TABLE statement for the target table.
- The database server does not clear out the contents of the violations table before or after it uses the violations table during an INSERT, UPDATE, DELETE, or SET Database Object Mode operation.
- If a target table has a filtering-mode constraint or unique index defined on it and a violations table associated with it, users cannot insert into the target table by selecting from the violations table. Before you insert rows into the target table by selecting from the violations table, you must take one of the following steps:
    - You can set the constraint or unique index to DISABLED mode.
    - You can issue STOP VIOLATIONS TABLE for the target table.

  If it is inconvenient to take either of these steps, but you intend to copy records from the violations table into the target table, a third option is to select from the violations table into a temporary table and then insert the contents of the temporary table into the target table.
- If the target table that is specified in the START VIOLATIONS TABLE statement is fragmented, the violations table has the same fragmentation strategy as the target table. Each fragment of the violations table is stored in the same dbspace partition as the corresponding fragment of the target table.
- Once a violations table is started for a target table, you cannot use the ALTER FRAGMENT statement to alter the fragmentation strategy of the target table or the violations table.
- If the target table specified in the START VIOLATIONS TABLE statement is not fragmented, the database server places the violations table in the same dbspace as the target table.
- If the target table has BYTE or TEXT columns, BYTE or TEXT data values in the violations table are created in the same blobspace that stores the BYTE or TEXT data in the target table.

Example of a Violations Table

To start a violations and diagnostics table for the target table named **customer** in the demonstration database, enter the following statement:

```
START VIOLATIONS TABLE FOR customer;
```

Because you include no USING clause, the violations table is named **customer_vio** by default. The **customer_vio** table includes these five columns:

Column One	Column Two	Column Three	Column Four	Column Five
customer_num fname lname	company address1 address2	city state	zipcode phone	informix_tupleid informix_optype informix_recowner

The **customer_vio** table has the same table definition as the **customer** table except that the **customer_vio** table has three additional columns that contain information about the operation that caused the nonconforming row.

Structure of the diagnostics table

When you issue a START VIOLATIONS TABLE statement for a target table, the diagnostics table that the statement creates has a predefined structure. This structure is independent of the structure of the target table.

The following table shows the schema of the diagnostics table.

Column Name	Data Type	Description
informix_tupleid	INTEGER	Implicitly refers to **informix_tupleid** column values in the violations table This relationship, however, is not declared as a foreign-key to primary-key relationship.
objtype	CHAR(1)	Identifies the type of violation This column can have the following values: C = Constraint violation I = Unique-index violation
objowner	CHAR(32)	Identifies the owner of the constraint or index for which an integrity violation was detected
objname	VARCHAR(128, 0)	Contains the name of the constraint or index for which an integrity violation was detected

Initial privileges on the diagnostics table

When the START VIOLATIONS TABLE statement creates the diagnostics table, the set of access privileges granted on the target table are a basis for granting privileges on the diagnostics table. The database server follows different rules, however, when it grants each type of privilege.

The following table explains the circumstances under which the database server grants each privilege on the diagnostics table.

Privilege

Condition for Granting the Privilege

Insert

User has the Insert privilege on the diagnostics table if the user has the Insert, Delete, or Update privilege on any column of the target table.

Delete

User has the Delete privilege on the diagnostics table if the user has the Insert, Delete, or Update privilege on any column of the target table.

Select

User has the Select privilege on the diagnostics table if the user has the Select privilege on any column in the target table.

Update

User has the Update privilege on the diagnostics table if the user has the Update privilege on any column in the target table.

Index

User has the Index privilege on the diagnostics table if the user has the Index privilege on the target table.

Alter

Alter privilege is not granted on the diagnostics table.

(Users cannot alter the schema of diagnostics tables.)

References

References privilege is not granted on the diagnostics table.

(Users cannot define referential constraints on diagnostics tables.)

The following rules concern access privileges on the diagnostics table:

- When the diagnostics table is created, the owner of the target table becomes the owner of the diagnostics table.
- The owner of the diagnostics table automatically receives all table-level privileges on the diagnostics table, including the Alter and References privileges. The database server, however, prevents the owner of the diagnostics table from altering the diagnostics table or adding a referential constraint to the diagnostics table.
- You can use the GRANT and REVOKE statements to modify the initial set of privileges on the diagnostics table.
- For INSERT, DELETE, or UPDATE operations on a target table that has a filtering-mode unique index or constraint defined on it, you must have the Insert privilege on the violations and diagnostics tables.

  If you do not have the Insert privilege on the violations and diagnostics tables, the database server executes the INSERT, DELETE, or UPDATE statement on the target table, provided you have the necessary privileges on the target table. The database server does not return an error concerning the lack of Insert privilege on the violations and diagnostics tables unless an integrity violation is detected during execution of the INSERT, DELETE, or UPDATE statement.

Similarly, when you issue a SET Database Object Mode statement to set a disabled constraint or disabled unique index to the enabled or filtering mode, and a violations table and diagnostics table exist for the target table, you must have the Insert privilege on the violations and diagnostics tables.

If you do not have the Insert privilege on the violations and diagnostics tables, the database server executes the SET Database Object Mode statement, provided you have the necessary privileges on the target table. The database server does not return an error concerning the lack of Insert privilege on the violations and diagnostics tables unless an integrity violation is detected during the execution of the SET Database Object Mode statement.

- The grantor of the initial set of privileges on the diagnostics table is the same as the grantor of the privileges on the target table. For example, if the user **jenny** was granted the Insert privilege on the target table by both the user **wayne** and the user **laurie**, both user **wayne** and user **laurie** grant the Insert privilege on the diagnostics table to user **jenny**.
- Once a diagnostics table is started for a target table, revoking a privilege on the target table from a user does not automatically revoke the same privilege on the diagnostics table from that user. Instead you must explicitly revoke the privilege on the diagnostics table from the user.
- If you have fragment-level privileges on the target table, you have the corresponding table-level privileges on the diagnostics table.

The next example illustrates how the initial set of privileges on a diagnostics table is derived from the current privileges on the target table. Assume that you have a table called **cust_subset** that holds **customer** data. This table consists of the following columns: **ssn** (social security number), **fname** (first name), **lname** (last name), and **city** (city in which the customer lives). The following set of access privileges exists on the **cust_subset** table:

- User **alvin** is the owner of the table.
- User **barbara** has the Insert and Index privileges on the table. She also has the Select privilege on the **ssn** and **lname** columns.
- User **danny** has the Alter privilege on the table.
- User **carrie** has the Update privilege on the **city** column. She also has the Select privilege on the **ssn** column.

Now user **alvin** starts a violations table named **cust_subset_viols** and a diagnostics table named **cust_subset_diags** for the **cust_subset** table:

```
START VIOLATIONS TABLE FOR cust_subset
   USING cust_subset_viols, cust_subset_diags;
```

The database server grants the following set of initial privileges on the **cust_subset_diags** diagnostics table:

- User **alvin** is the owner of the diagnostics table, so he has all table-level privileges on the table.
- User **barbara** has the Insert, Delete, Select, and Index privileges on the diagnostics table.
- User **carrie** has the Insert, Delete, Select, and Update privileges on the diagnostics table.
- User **danny** has no privileges on the diagnostics table.

Using the Diagnostics Table

The following issues concern the structure and use of the diagnostics table.

- The MAX ROWS clause of the START VIOLATIONS TABLE statement sets a limit on the number of rows that can be inserted into the diagnostics table when you execute a single statement, such as an INSERT or SET Database Object Mode statement, on the target table.
- The MAX ROWS clause limits the number of rows only for operations in which the table functions as a diagnostics table.
- When a table functions as a diagnostics table, it cannot have triggers or constraints defined on it.
- When a table functions as a diagnostics table, users can create indexes on the table, but the existence of an index affects performance. You cannot set unique indexes on a diagnostics table to FILTERING database object mode.
- If a target table has a violations and diagnostics table associated with it, dropping the target table in cascade mode (the default mode) causes the violations and diagnostics tables to be dropped also.
- If the target table is dropped in restricted mode, the DROP TABLE operation fails (because the violations and diagnostics tables exist).
- Once a violations table is started for a target table, you cannot use the ALTER TABLE statement to add, modify, or drop columns in the target table, violations table, or diagnostics table. Before you can alter any of these tables, you must issue a STOP VIOLATIONS TABLE statement for the target table.
- The database server does not clear out the contents of the diagnostics table before or after it uses the diagnostics table during an Insert, Update, Delete, Merge, SET CONSTRAINTS, or SET INDEXES operation.
- If the target table that is specified in the START VIOLATIONS TABLE statement is fragmented, the diagnostics table is fragmented with a round-robin strategy over the same dbspaces in which the target table is fragmented.

To start a violations and diagnostics table for the target table named **stock** in the demonstration database, enter the following statement:

```
START VIOLATIONS TABLE FOR stock;
```

Because your START VIOLATIONS TABLE statement does not include a USING clause, the diagnostics table is named **stock_dia** by default. The **stock_dia** table includes the following two columns:

Column One	Column Two
informix_tupleid objtype	objowner objname

This list of columns shows an important difference between the diagnostics table and violations table for a target table. Whereas the violations table has a matching column for every column in the target table, the columns of the diagnostics table do not match any columns in the target table. The diagnostics table created by any START VIOLATIONS TABLE statement always has the same columns, with the same column names and data types.

For information on the relationship between the diagnostics table and the violations table, see Relationships Among the Target, Violations, and Diagnostics Tables on page 848.

STOP VIOLATIONS TABLE statement

Use the STOP VIOLATIONS TABLE statement to drop the association between a target table, its violations table, and its diagnostics table. This statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
STOP VIOLATIONS TABLE FOR [ 'owner' . ] table
```

Element	Description	Restrictions	Syntax
owner	The owner of *table*	Must own the table	Owner name on page 1262
table	Name of a target table whose association with the violations and diagnostics table is to be dropped. No default value exists.	Must be a local table that has an associated violations table and a diagnostics table	Identifier on page 1231

Usage

The STOP VIOLATIONS TABLE statement drops the association between the target table, the violations table, and the diagnostics table. After you issue this statement, the former violations and diagnostics tables continue to exist, but they no longer function as violations and diagnostics tables for the target table. They now have the status of regular database tables instead of violations and diagnostics tables for the target table. You must issue the DROP TABLE statement to drop these two tables explicitly.

When DML operations (INSERT, DELETE, or UPDATE) cause data-integrity violations for rows of the target table, the nonconforming rows are no longer filtered to the former violations table, and diagnostic information about the data-integrity violations is not placed in the former diagnostics table.

Example of Stopping the Violations and Diagnostics Tables

Assume that a target table named **cust_subset** has an associated violations table named **cust_subset_vio** and an associated diagnostics table named **cust_subset_dia**. To drop the association between the target table and the violations and diagnostics tables, enter the following statement:

```
STOP VIOLATIONS TABLE FOR cust_subset;
```

This deletes the row in the **sysviolations** system catalog table that had registered the former association. Subsequent DML operations on the target **cust_subset** table will no longer cause the database server to insert information about nonconforming rows into its former violations table or diagnostics table.

Example of Dropping the Violations and Diagnostics Tables

After you execute the STOP VIOLATIONS TABLE statement in the preceding example, the **cust_subset_vio** and the **cust_subset_dia** tables continue to exist, but they are no longer associated with the **cust_subset** table. Instead they now have the status of regular database tables. To drop these two tables, enter the following statements:

```
DROP TABLE cust_subset_vio;
DROP TABLE cust_subset_dia;
```

If you had previously issued the DROP TABLE statement without the RESTRICT keyword to successfully drop the **cust_subset** table, then the statements above would not be necessary, because dropping the target table in cascade mode implicitly drops any associated violations table and diagnostics table.

Privileges Required for Stopping a Violations Table

To stop a violations or a diagnostics table for a given target table, you must meet one of the following requirements:

- You must have the DBA privilege on the database.
- You must be the owner of the target table and have the Resource privilege on the database.
- You must have the Alter privilege on the target table and the Resource privilege on the database.

TRUNCATE statement

Use the TRUNCATE statement to quickly delete all rows from a local table and free the associated storage space. You can optionally reserve the space for the same table and its indexes. Only Informix® supports this implementation of the TRUNCATE statement, which is an extension to the ANSI/ISO standard for SQL.

Syntax

```
              .- TABLE -.
>>- TRUNCATE -+---------+--+---------------+--+- table ---+-------->
                        '- 'owner' -- . --'  '- synonym -'

   .- DROP STORAGE --.
>--+-----------------+- -+-------------------+----------------><
   '- REUSE STORAGE -'   '- KEEP STATISTICS -'
```

Element	Description	Restrictions	Syntax
owner	Owner of *table* or *synonym*	See **Usage** notes.	Owner name on page 1262
synonym	Synonym for the table from which to remove all data	Must exist, and USETABLENAME must not be set	Identifier on page 1231
table	Name of table from which to remove all data and all B-tree structures of its indexes	Must exist in the database	Identifier on page 1231

Usage

The TRUNCATE statement rapidly deletes from a local table all active data rows and the B-tree structures of indexes on the table. You have the option of releasing the storage space that was occupied by the rows and index extents, or of reusing the same space when the table is subsequently repopulated with new rows.

To execute the TRUNCATE statement, at least one of the following conditions must be satisfied:

- You are the owner of the table.
- You hold the Delete access privilege on the table.
- You hold the DBA access privilege on the current database.

If an enabled Delete trigger is defined on the table, you must also hold the Alter privilege on the table, even though the TRUNCATE statement does not activate triggers.

Although it requires the Delete privilege for a non-DBA user who does not own the table, TRUNCATE is a data definition language (DDL) statement. Like other DDL statements, TRUNCATE cannot operate on any table outside the database to which you are connected, nor on a table that a concurrent session is reading in Dirty Read isolation mode.

Informix® always logs the TRUNCATE operation, even for a non-logging table. In databases that support transaction logging, only the COMMIT WORK or ROLLBACK WORK statement of SQL is valid after the TRUNCATE statement within the same transaction. Here the ROLLBACK statement must cancel the entire transaction that includes the TRUNCATE statement. Informix® issues an error if ROLLBACK TO SAVEPOINT (or any other SQL statement except for COMMIT WORK or ROLLBACK WORK without the TO SAVEPOINT clause) immediately follows the TRUNCATE statement.

When you successfully rollback the TRUNCATE statement, no rows are removed from the table, and the storage extents that hold the rows and index partitions continue to be allocated to the table. Only databases with transaction logging can support the ROLLBACK WORK statement.

After the TRUNCATE statement successfully executes, Informix® automatically updates the statistics and distributions for the table and for its indexes in the system catalog to show no rows in the table nor in its dbspace partitions. It is not necessary to run the UPDATE STATISTICS statement immediately after you commit the TRUNCATE statement. If the KEEP STATISTICS operation is used, then TRUNCATE TABLE does not update the system catalog.

If the table that the TRUNCATE statement specifies is a typed table, a successful TRUNCATE operation removes all the rows and B-tree structures from that table and from all its subtables within the table hierarchy.

The TRUNCATE statement does not reset the serial value of SERIAL, BIGSERIAL, or SERIAL8 columns. To reset the counter of a serial column, use the MODIFY clause of the ALTER TABLE statement, either before or after you execute the TRUNCATE statement.

The TABLE Keyword

The TABLE keyword has no effect on this statement, but it can be included to make your code more legible for human readers. Both of the following statements have the same effect, deleting all rows and any related index data from the **customer** table:

```
TRUNCATE TABLE customer;

TRUNCATE customer;
```

The Table Specification

You must specify the name or synonym of a table in the local database to which you are currently connected. If the **USETABLENAME** environment variable is set, you must use the name of the table, rather than a synonym. The table can be of type STANDARD, RAW, or TEMP, but you cannot specify the name or synonym of a view, or an object that the CREATE EXTERNAL TABLE statement defined. (Categories of tables that are not valid with TRUNCATE are listed in the section Restrictions on the TRUNCATE statement on page 862.)

In a database that is ANSI-compliant, you must specify the *owner* qualifier if you are not the owner of the *table* or *synonym*.

After the TRUNCATE statement begins execution, Informix® attempts to place an exclusive lock on the specified table, to prevent other sessions from locking the table until the TRUNCATE statement is committed or rolled back. Exclusive locks are also applied to any dependent tables of the truncated table within a table hierarchy.

While concurrent sessions that use the Dirty Read isolation level are reading the table, however, the TRUNCATE statement fails with an RSAM -106 error. To reduce this risk, you can set the **IFX_DIRTY_WAIT** environment variable to specify that the TRUNCATE operation wait for a specified number of seconds for Dirty Read transactions to commit or rollback.

The STORAGE specification

The optional DROP STORAGE or REUSE STORAGE keywords specify what action the database server takes with the storage extents of the table when the TRUNCATE operation begins. If you omit this specification, the DROP STORAGE option is the default.

Using the default or explicit DROP STORAGE option, a successful TRUNCATE statement releases all but the first extent among the extents currently allocated to the table and to its indexes. You can display the current list of extents with the `oncheck -pT` *table* command. If your table has only one extent, no space will be freed.

For the following table, for example, four default eight-page extents were merged into what is now the first extent. There is also a second, larger extent:

```
Extents
   Logical Page  Physical Page     Size Physical Pages
              0       1:104455       32              32
             32       1:104495     4576            4576
```

After the TRUNCATE statement runs, output from the same `oncheck` command displays this:

```
Extents
   Logical Page  Physical Page     Size Physical Pages
              0       1:104455       32              32
```

Alternatively, if you intend to keep the same storage space allocated to the same table for subsequently loaded data, specify the REUSE STORAGE keywords to prevent the space from being deallocated. The REUSE STORAGE option of TRUNCATE can make storage management more efficient in applications where the same table is periodically emptied and reloaded with new rows.

The following example truncates the **state** table and frees all of its extents except the first extent:

```
TRUNCATE TABLE state DROP STORAGE;
```

The following example truncates the same table but removes only the actual data. All extents stay the same.

```
TRUNCATE TABLE state REUSE STORAGE;
```

Whether you specify DROP STORAGE or REUSE STORAGE, any out-of-row data values are released for all rows of the table when the TRUNCATE transaction is committed. Storage occupied by any BLOB or CLOB values that become unreferenced in the TRUNCATE transaction is also released.

The AM_TRUNCATE Purpose Function

Informix® provides built-in **am_truncate** purpose functions for its primary access methods that support TRUNCATE operations on columns of permanent and temporary tables. It also provides a built-in **am_truncate** purpose function for its secondary access method for TRUNCATE operations on B-tree indexes.

For the TRUNCATE statement to work correctly in a virtual table interface (VTI) table requires a valid **am_truncate** purpose function in the primary access method for the data type of the VTI table. To register a new primary access method in the database, use the CREATE PRIMARY ACCESS_METHOD statement of SQL:

```
CREATE PRIMARY ACCESS_METHOD vti(
  AM_GETNEXT  = vti_getnext
  AM_TRUNCATE = vti_truncate
  ...);
```

You can also use the ALTER ACCESS_METHOD statement to add a valid **am_truncate** purpose function to an existing access method that has no **am_truncate** purpose function:

```
ALTER ACCESS_METHOD abc (ADD AM_TRUNCATE = abc_truncate);
```

In these examples, the **vti_truncate** and **abc_truncate** functions must be routines that support the functionality of the AM_TRUNCATE purpose option keyword, and that were previously registered in the database by the CREATE FUNCTION or CREATE ROUTINE FROM statement.

Performance Advantages of TRUNCATE

The TRUNCATE statement is not equivalent to DROP TABLE. After TRUNCATE successfully executes, the specified *table* and all its columns and indexes are still registered in the database, but with no rows of data. In information management applications that require replacing all of the records in a table after some time interval, TRUNCATE requires fewer updates to the system catalog than the equivalent DROP TABLE, CREATE TABLE, and any additional DDL statements to redefine any synonyms, views, constraints, triggers, privileges, fragmentation schemes, and other attributes and associated database objects of the table.

In contexts where no existing rows of a table are needed, the TRUNCATE statement is typically far more efficient than using the DELETE statement with no WHERE clause to empty the table, because TRUNCATE requires fewer resources and less logging overhead than DELETE:

- DELETE FROM *table* deletes each row as a separately logged operation. If indexes exist on the table, each index must be updated when a row is deleted, and this update is also logged for each row. If an enabled Delete trigger is defined on the table, its triggered actions must also be executed and logged.
- TRUNCATE *table* performs the removal of all rows and of the B-tree structures of every index on the table as a single operation, and writes a single entry in the logical log when the transaction that includes TRUNCATE is committed or rolled back. The triggered action of any enabled trigger is ignored.

These performance advantages of TRUNCATE over DELETE are reduced when the table has one or more columns with the following attributes:

- Any simple large object data types stored in blobspaces
- Any BLOB, CLOB, complex, or user-defined types stored in sbspaces
- Any opaque types for which a **destroy** support function is defined.

Each of these features require the database server to read each row of the table, substantially reducing the speed of TRUNCATE.

If a table includes one or more UDTs for which you have registered an **am_truncate( )** purpose function, then the performance difference between TRUNCATE and DELETE would reflect the relative costs of invoking the **am_truncate** interface once for TRUNCATE versus invoking the **destroy( )** support function for each row.

As listed in the next section, certain conditions cause TRUNCATE to fail with an error. Some of these conditions have no effect on DELETE operations, so in those cases you can remove all rows more efficiently with a DELETE statement, as in the following operation on the **customer** table:

```
DELETE customer;
```

The FROM keyword that immediately follows DELETE can be omitted, as in this example, only if the **DELIMIDENT** environment variable is set.

Restrictions on the TRUNCATE statement

The TRUNCATE statement fails if any of the following conditions exist:

- The user does not hold the Delete access privilege on the table.
- The table has an enabled Delete trigger, but the user lacks the Alter privilege.
- The specified table or synonym does not exist in the local database.
- The table was defined by the CREATE EXTERNAL TABLE statement.
- The specified synonym does not reference a table in the local database.
- The statement specifies a synonym for a local table, but the **USETABLENAME** environment variable is set.
- The statement specifies the name of a view or a synonym for a view.
- The table is a system catalog table or a system-monitoring interface (SMI) table.
- An R-tree index is defined on the table.
- The table is a virtual table (or has a virtual-index interface) for which no valid **am_truncate** access method exists in the database.
- An Enterprise Replication replicate that is not a master replicate is defined on the table. (For more information about replicates, see the .)
- A shared or exclusive lock on the table already exists.
- One or more cursors are open on the table.
- A concurrent session with Dirty Read isolation level is reading the table.
- Another table, with at least one row, has an enabled foreign-key constraint on the specified table. (An enabled foreign key constraint of another table that has no rows, however, has no effect on a TRUNCATE operation.)

UNLOAD statement

Use the UNLOAD statement to write the rows retrieved by a SELECT statement to an operating-system file. The UNLOAD statement is an extension to the ANSI/ISO standard for SQL.

Syntax

Only DB-Access supports the UNLOAD statement.

```
UNLOAD TO 'filename' [ DELIMITER 'delimiter' ] { <SELECT Statement> (explicit id ) | variable }
```

Element	Description	Restrictions	Syntax
delimiter	Quoted string to specify the field delimiter character in *filename* file	See DELIMITER Clause on page 867	Quoted String on page 1202
filename	Operating-system file to receive the rows. Default *pathname* is the current directory.	See UNLOAD TO File on page 863.	Quoted String on page 1202
variable	Host variable that contains the text of a valid SELECT statement	Must have been declared as a character data type	Language- specific

Usage

Important: Use the UNLOAD statement only with .

The UNLOAD statement copies to a file the rows retrieved by a query. You must have the Select privilege on all columns specified in the SELECT statement. For information on database-level and table-level privileges, see GRANT statement on page 537.

You can specify a literal SELECT statement or a character variable that contains the text of a SELECT statement. (See SELECT statement on page 690.)

The following example unloads rows whose value of `customer.customer_num` is greater than or equal to `138`, and writes them to a file named `cust_file`:

```
UNLOAD TO 'cust_file' DELIMITER '!'
   SELECT * FROM customer WHERE customer_num> = 138;
```

The resulting output file, `cust_file`, contains two rows of data values:

```
138!Jeffery!Padgett!Wheel Thrills!3450 El Camino!Suite 10!Palo Alto!CA!94306!!
139!Linda!Lane!Palo Alto Bicycles!2344 University!!Palo Alto!CA!94301!
    (415)323-5400
```

UNLOAD TO File

The UNLOAD TO file, as specified by the *filename* parameter, receives the retrieved rows.

You can use an UNLOAD TO file as input to a LOAD statement.

UNLOAD TO data formats in the default U.S. English locale

In the default locale, data values have these formats in the UNLOAD TO file.

Data Type	Output Format
BOOLEAN	BOOLEAN values appear as either t for TRUE or f for FALSE.
Character	If a character field contains the delimiter, HCL® Informix® products automatically escape it with a backslash (`\`) to prevent interpretation as a special character. In the UNLOAD TO file, a literal backslash character is represented as two consecutive backslash characters (`\\`). If you use the LOAD statement of DB-Access to insert the rows into a table, the backslash escape characters are automatically stripped from that table.
Collections	A collection is unloaded with its values between braces (`{ }`) and a delimiter between each element. For more information, see Unloading Complex Types on page 866.
DATE	DATE values are represented as *mm*/*dd*/*yyyy* (or the default format for the database locale), where *mm* is the month (January = `1`, and so on), *dd* is the day, and *yyyy* is the year. If you have set the **GL_DATE** or **DBDATE** environment variable, the UNLOAD statement uses the specified date format for DATE values.
DATETIME, INTERVAL	Literal DATETIME and INTERVAL values appear as digits and delimiters, without keyword qualifiers, in the default format *yyyy-mm-dd hh:mi:ss.fff*. Time units outside the declared precision are omitted. If the **GL_DATETIME** or **DBTIME** environment variable is set, DATETIME values appear in the specified format.
DECIMAL, MONEY	Values are unloaded with no leading currency symbol. In the default locale, comma (,) is the *thousands* separator and period (.) is the decimal separator. If **DBMONEY** is set, UNLOAD uses its specified separators and currency format for MONEY values.
NULL	NULL appears as two delimiters with no characters between them.
Number	Values appear as literals, with no leading blanks. For BIGINT, INTEGER, INT8, and SMALLINT data types, zero appears as `0`, For MONEY, FLOAT, SMALLFLOAT, and DECIMAL data types, zero appears as `0.0`.
ROW types (named and unnamed)	A ROW type is unloaded with its values enclosed between parentheses and a field delimiter separating each element. For more information, see Unloading Complex Types on page 866.
Simple large objects (TEXT, BYTE)	TEXT and BYTE columns are unloaded directly into the UNLOAD TO file. BYTE values appear in ASCII hexadecimal form, with no added white space or newline characters. For more information, see Unloading Simple Large Objects on page 865.
Smart large objects (CLOB, BLOB)	CLOB and BLOB columns are unloaded into a separate operating-system file (for each column) on the client computer. The CLOB or BLOB field in the UNLOAD TO file contains the name of this file. For more information, see Unloading Smart Large Objects on page 865.

Data Type	Output Format
User-defined data types (opaque types)	Opaque types must have an **export( )** support function defined. They need special processing to copy data from the internal format of the opaque data type to the UNLOAD TO file format. An **exportbinary( )** support function might also be required for data in binary format. The data in the UNLOAD TO file would correspond to the format that the **export( )** or **exportbinary( )** support function returns.

UNLOAD TO data formats in nondefault locales

In nondefault locales, DATE, DATETIME, MONEY, and numeric column values have formats that the locale supports for these data types. For more information, see the *HCL® Informix® GLS User's Guide*. For more information on **DBDATE**, **DBMONEY**, and **DBTIME** environment variables, refer to the *HCL® Informix® Guide to SQL: Reference*.

In databases that use a nondefault locale, if the **GL_DATETIME** environment variable has a nondefault setting, the **USE_DTENV** environment variable must be set to `1` before the UNLOAD statement can correctly unload localized DATETIME values from the database table, or from view, or from a table object defined by the EXTERNAL TABLE statement. For more information on the **GL_DATETIME**, **GL_DATE**, and **USE_DTENV** environment variables, refer to the *HCL® Informix® GLS User's Guide*.

Unloading Simple Large Objects

The database server writes BYTE and TEXT values directly into the UNLOAD TO file. BYTE values are written in hexadecimal dump format with no added blank spaces or new line characters. The logical length of an UNLOAD TO file containing BYTE data can therefore be long and difficult to print or edit.

If you are unloading files that contain simple-large-object data types, do not use characters that can appear in BYTE or TEXT values as delimiters in the UNLOAD TO file. See also the section DELIMITER Clause on page 867.

The database server handles any required code-set conversions for TEXT data. For more information, see the *HCL® Informix® GLS User's Guide*.

If you are unloading files that contain simple-large-object data types, objects smaller than 10 kilobytes are stored temporarily in memory. You can adjust the 10-kilobyte setting to a larger setting with the **DBBLOBBUF** environment variable. BYTE or TEXT values larger than the default or the **DBBLOBBUF** setting are stored in a temporary file. For additional information about **DBBLOBBUF**, see the *HCL® Informix® Guide to SQL: Reference*.

Unloading Smart Large Objects

The database server unloads smart large objects (from BLOB or CLOB columns) into a separate operating-system file for each column, in the same directory on the client computer as the UNLOAD TO file. All the smart blobs in the same column are stored in a single file. The filename has one of these formats:

- For a BLOB value: `blob########`
- For a CLOB value: `clob########`

In the preceding formats, the pound (#) symbols represent the digits of the unique hexadecimal smart-large-object identifier of the first smart large object in the file. The maximum number of digits for a smart-large-object identifier is 17. Most smart large objects, however, would have an identifier with fewer digits.

When the database server unloads the first smart large object, it creates the appropriate BLOB or CLOB client file with the hexadecimal identifier of the smart large object. If additional smart-large-object values are present in the same column, the database server writes their values to the same file, and lists in the UNLOAD TO file (***.unl**) the sbspace, chunk, and page numbers, and smart large object identifier, for each BLOB or CLOB value in the file.

The following example shows an UNLOAD TO file entry for two smart large object values from the same column:

```
Object # 1
Space Chunk Page = [5,6,3] Object ID = 1192071051

Object #2
Space Chunk Page = [5,6,4] Object ID = 1192071050

both rows unloaded
```

In an UNLOAD TO file, a BLOB or CLOB column value appears in this format:

```
start_off,length,client_path
```

Here *start_off* is the starting offset (in hexadecimal format) of the smart-large-object value within the client file, *length* is the length (in hexadecimal) of the BLOB or CLOB value, and *client_path* is the pathname for the client file. No blank spaces can appear between these values. If a CLOB value is 512 bytes long and is at offset 256 in the **/usr/apps/clob9ce7.318** file, for example, then the CLOB value appears as follows in the UNLOAD TO file:

```
|100,200,/usr/apps/clob9ce7.318|
```

If a BLOB or CLOB column value occupies an entire client file, the CLOB or BLOB column value appears as follows in the UNLOAD TO file:

```
client_path
```

For example, if a CLOB value occupies the entire file **/usr/apps/clob9ce7.318**, the CLOB value appears as follows in the UNLOAD TO file:

```
|/usr/apps/clob9ce7.318|
```

For locales that support multibyte code sets, be sure that the declared size (in bytes) of any column that receives character data is large enough to store the entire data string. For some locales, this can require up to 4 times the number of logical characters in the longest data string.

The database server handles any required code-set conversions for CLOB data. For more information, see the *HCL® Informix® GLS User's Guide*.

Unloading Complex Types

In an UNLOAD TO file, values of complex data types appear as follows:

- Collections are introduced with the appropriate constructor (MULTISET, LIST, SET), with their comma-separated elements enclosed in braces ({ }):

  ```
  constructor{val1 , val2 , ... }
  ```

  For example, to unload the SET values {1, 3, 4} from a column of the SET (INTEGER NOT NULL) data type, the corresponding field of the UNLOAD TO file appears as follows:

  ```
  |SET{1 , 3 , 4}|
  ```

- ROW types (named and unnamed) are introduced by the ROW constructor and have their fields enclosed between parentheses and comma-separated:

  ```
  ROW(val1 , val2 , ... )
  ```

  For example, to unload the ROW values (`1, 'abc'`), the corresponding field of the UNLOAD TO file appears as follows:

  ```
  |ROW(1 , abc)|
  ```

DELIMITER Clause

Use the DELIMITER clause of the UNLOAD statement to specify the delimiter that separates the data contained in each column in a row in the output file.

If you omit this clause, then checks the setting of the **DBDELIMITER** environment variable. If **DBDELIMITER** has not been set, the default delimiter is the pipe (|) symbol. You can specify the tab character `TAB` (`CTRL-I`) or a blank space (ASCII 32) as the delimiter symbol.

The following characters are not valid in any locale as delimiter symbols:

- Newline character (`CTRL-J`)
- Hexadecimal digits (`0` to `9`, `a` to `f`, `A` to `F`)
- Backslash (\ )

The backslash (\ ) is not a valid field separator or record delimiter because it is the default escape character, indicating that the next character is a literal character in the data, rather than a special character. However, if you change the default escape character by setting the DEFAULTESCCHAR configuration parameter or the **DEFAULTESCCHAR** session environment option, you can use a backslash as a field separator.

The following UNLOAD statement specifies the semicolon (;) as the delimiter:

```
UNLOAD TO 'cust.out' DELIMITER ';'
   SELECT fname, lname, company, city FROM customer;
```

UNLOCK TABLE statement

Use the UNLOCK TABLE statement in a database that does not support transaction logging to unlock a table that you previously locked with the LOCK TABLE statement. The UNLOCK TABLE statement is an extension to the ANSI/ISO standard for SQL.

Syntax

```
UNLOCK TABLE { table | synonym }
```

Element	Description	Restrictions	Syntax
synonym	Synonym for a table to unlock	The synonym and the table to which it points must exist	Database Object Name on page 1225
table	Table to unlock	Must be in a database without transaction logging, and must be a table that you previously locked	Database Object Name on page 1225

Usage

Restriction: The UNLOCK TABLE statement is not valid within a transaction.

You can lock a table if you own the table or if you have the Select privilege on the table, either from a direct grant to your user ID or from a grant to PUBLIC. You can only unlock a table that you locked. You cannot unlock a table that another process locked. Only one lock can apply to a table at a time.

You must specify the name or synonym of the table that you are unlocking. Do not specify the name of a view, or a synonym for a view.

To change the lock mode of a table in a database that was created without transaction logging, use the UNLOCK TABLE statement to unlock the table, then issue a new LOCK TABLE statement. The following example shows how to change the lock mode of a table in a database that was created without transaction logging:

```
LOCK TABLE items IN EXCLUSIVE MODE;
...
UNLOCK TABLE items;
...
LOCK TABLE items IN SHARE MODE;
```

The UNLOCK TABLE statement fails if it is issued within a transaction. Table locks set within a transaction are released automatically when the transaction completes.

If you are using an ANSI-compliant database, do not issue an UNLOCK TABLE statement. The UNLOCK TABLE statement fails if it is issued within a transaction, and a transaction is always in effect in an ANSI-compliant database.

UPDATE statement

Use the UPDATE statement to change the values in one or more columns of one or more existing rows in a table or view.

Syntax

(explicit id unique_560_Connect_42_upd001) unique_560_Connect_42_upd001(explicit id unique_560_Connect_42_upd004)
unique_560_Connect_42_upd004(explicit id unique_560_Connect_42_upd006) unique_560_Connect_42_upd006(explicit id unique_560_Connect_42_upd008)
unique_560_Connect_42_upd008(explicit id unique_560_Connect_42_upd009) unique_560_Connect_42_upd009(explicit id unique_560_Connect_42_upd010)
unique_560_Connect_42_upd010(explicit id unique_560_Connect_42_upd011) unique_560_Connect_42_upd011

```
UPDATE { [ <Optimizer Directives> ]
<Target> <SET Clause> <WHERE Options> | <Collection-Derived Table> <SET Clause> [ WHERE CURRENT OF cursor ] }
```

Target

```
" { { table | view | synonym } [ AS ] alias ] | { | ONLY ( { table | synonym } ) } } "
```

WHERE Options

```
" { [ WHERE condition ] | WHERE CURRENT OF cursor_id } "
```

Element	Description	Restrictions	Syntax
alias	Temporary name that you declare here for a local or remote table	The AS keyword must precede *alias* if SET is the identifier of *alias*	Identifier on page 1231
condition	Logical criteria that updated rows must satisfy	Cannot be a UDR nor a correlated subquery	Condition on page 967
cursor	Name of a cursor whose current row is to be updated	Cannot be a host variable. You cannot update a row that includes aggregates	Identifier on page 1231
synonym, *table*, *view*	Synonym, table, or view that contains rows to be updated	The *synonym* and the table or view to which it points must exist	Database Object Name on page 1225

Usage

Use the UPDATE statement to update any of the following types of database objects or program objects:

- A row in a table: a single row, a group of rows, or all rows in a table
- An element in a column of a collection data type
- In a column of a named or unnamed ROW data type, a field or all fields.

With Informix®, you can also use this statement to change the values in one or more elements in or SPL collection variables or ROW variables.

For information on how to update elements of a collection variable, see Collection-Derived Table on page 1212. Sections that follow in this description of the UPDATE statement describe how to update a row in a table.

You must either own the table or have the Update privilege for the table; see GRANT statement on page 537. To update data in a view, you must have the Update privilege, and the view must meet the requirements that are explained in Updating Rows Through a View on page 870.

The target of the UPDATE statement cannot be a table object that the CREATE EXTERNAL TABLE statement defined.

The cursor (as defined in the SELECT ... FOR UPDATE portion of a DECLARE statement) can contain only column names. If you omit the WHERE clause, all rows of the target table are updated.

If you are using effective checking and the checking mode is set to IMMEDIATE, all enabled constraints are checked at the end of each UPDATE statement. If the checking mode is set to DEFERRED, all enabled constraints are *not* checked until the transaction is committed.

In DB-Access, if you omit the WHERE clause and are in interactive mode, does not run the UPDATE statement until you confirm that you want to change all rows. If the statement is in a command file, however, and you are running at the command line, the statement executes immediately.

Example

The following example creates and updates view.

```
CREATE VIEW cust_view AS SELECT * FROM customer;
UPDATE cust_view SET customer_num=10001 WHERE customer_num=101;
```

Using the ONLY Keyword

If you use the UPDATE statement to update rows of a supertable, rows from its subtables can also be updated. To update rows from the supertable only, use the ONLY keyword prior to the table name, as this example shows:

```
UPDATE ONLY(am_studies_super)
   WHERE advisor = "johnson"
      SET advisor = "camarillo";
```

Note: If you use the UPDATE statement on a supertable without the ONLY keyword and without a WHERE clause, all rows of the supertable and its subtables are updated. You cannot use the ONLY keyword if you plan to use the WHERE CURRENT OF clause to update the current row of the active set of a cursor.

Updating Rows Through a View

You can update data through a *single-table* view if you have the Update privilege on the view (see GRANT statement on page 537). For a view to be updatable, the query that defines the view must not contain any of the following items:

- Columns in the projection list that are aggregate values
- Columns in the projection list that use the UNIQUE or DISTINCT keyword
- A GROUP BY clause
- A UNION operator

In addition, if a view is built on a table that has a derived value for a column, that column cannot be updated through the view. Other columns in the view, however, can be updated. In an updatable view, you can update the values in the underlying table by inserting values into the view.

```
CREATE VIEW cust_view AS SELECT * FROM customer;
UPDATE cust_view SET customer_num=10001 WHERE customer_cum=101;
```

The following statements define a view that includes all the rows in the **customer** table and changes the **customer_num** value to `10001` in any row where the value of that column is `101`:

```
CREATE VIEW cust_view AS SELECT * FROM customer;
UPDATE cust_view SET customer_num=10001 WHERE customer_num=101;
```

You can use data-integrity constraints to prevent users from updating values in the underlying table when the update values do not fit the SELECT statement that defined the view. For more information, see WITH CHECK OPTION Keywords on page 425.

Because duplicate rows can occur in a view even if its base table has unique rows, be careful when you update a table through a view. For example, if a view is defined on the **items** table and contains only the **order_num** and **total_price** columns, and if two items from the same order have the same total price, the view contains duplicate rows. In this case, if you update one of the two duplicate **total_price** values, you have no way to know which item price is updated.



Important: If you are using a view with a check option, you cannot update rows in a remote table.

An alternative to directly modifying data values in a view with the UPDATE statement is to create an INSTEAD OF trigger on the view. For more information, see #unique_516.

Updating Rows in a Database Without Transactions

If you are updating rows in a database without transactions, you must take explicit action to restore updated rows. For example, if the UPDATE statement fails after updating some rows, the successfully updated rows remain in the table. You cannot automatically recover from a failed update.

Updating Rows in a Database with Transactions

If you are updating rows in a database with transactions, and you are using transactions, you can undo the update using the ROLLBACK WORK statement. If you do not execute a BEGIN WORK statement before the update, and the update fails, the database server automatically rolls back any database modifications that were made to the table since the beginning of the update operation.

The CREATE TEMP TABLE statement can include the WITH NO LOG option to create temporary tables that do not support transaction logging. These tables are not logged and are not recoverable.

Do not use RAW tables within a transaction. Tables that you create with the CREATE RAW TABLE statement are not logged. Thus, RAW tables are not recoverable, even if the database uses logging. RAW tables are intended for the initial loading and validation of data. After you have loaded the data, use the ALTER TABLE statement to change the table to type STANDARD,

and perform a level-0 backup before you use the table in a transaction. For more information about RAW tables, refer to the *HCL® Informix® Administrator's Guide*.

In an ANSI-compliant database, transactions are implicit, and all database modifications take place within a transaction. In this case, if an UPDATE statement fails, you can use ROLLBACK WORK to undo the update.

If you are within an explicit transaction, and the update fails, the database server automatically undoes the effects of the update.

Locking Considerations

When a row is selected with the intent to update, the update process acquires an update lock. Update locks permit other processes to read, or *share*, a row that is about to be updated, but they do not allow those processes to update or delete it.

Just before the update occurs, the update process *promotes* the shared lock to an exclusive lock. An exclusive lock prevents other processes from reading or modifying the contents of the row until the lock is released.

An update process can acquire an update lock on a row or on a page that has a shared lock from another process, but you cannot promote the update lock from shared to exclusive (and the update cannot occur) until the other process releases its lock.

If the number of rows that a single update affects is large, you can exceed the limits placed on the maximum number of simultaneous locks. If this occurs, you can reduce the number of transactions per UPDATE statement, or you can lock the page or the entire table before you execute the statement.

Declaring an alias for the target table

You can declare an alias for the target table. The alias can reference the fully-qualified database object name of a local or remote table, view, or synonym.

The alias is a temporary name that is not registered in the system catalog of the database, and that persists only while the UPDATE statement is running.

If the name that you declare as the alias is also a keyword of the UPDATE statement, you must use the AS keyword to clarify the syntax:

```
UPDATE stock AS set
   SET unit_price = unit_price * 0.94;
```

The following UPDATE statement references the qualified name of a table in the target clause and in two subqueries:

```
UPDATE nmosdb@wnmserver1:test
SET name=(SELECT name FROM test
   WHERE test.id = nmosdb@wnmserver1:test.id)
WHERE EXISTS(
SELECT 1 FROM test WHERE test.id = nmosdb@wnmserver1:test.id
);
```

The next UPDATE statement is logically equivalent to the previous example, but declares **r_t** an alias for the qualified table name:

```
UPDATE nmosdb@wnmserver1:test r_t
SET name=(SELECT name FROM test
   WHERE test.id = r_t.id)
WHERE EXISTS(
   SELECT 1 FROM test WHERE test.id = r_t.id
);
```

Declaring the table alias simplifies the notation of the second example above.

SET Clause

Use the SET clause to identify the columns to be updated and assign values to each column.

The SET clause supports the following syntax formats:

- A single-column format, which pairs each column with a single expression
- A multiple-column format, which associates a list of multiple columns with the values returned by one or more expressions

(explicit id unique_863_Connect_42_setcl001) unique_863_Connect_42_setcl001(explicit id unique_863_Connect_42_setcl002)

unique_863_Connect_42_setcl002(explicit id unique_863_Connect_42_setcl003) unique_863_Connect_42_setcl003

SET Clause

> " SET "

> " { <Single-Column Format> [] | [] <Multiple-Column Format> [] } "

Example

The following UPDATE statement uses the single-column format of the SET clause to change the value of the **customer.city** column of any existing row whose value in the **customer.customer_num** column is `103`.

```
UPDATE customer
   SET city = 'Alviso'
      WHERE customer_num = 103;
```

If no row has `103` as its **customer_num** column value, this example has no effect on the **customer** table.

Single-Column Format

Use the single-column format to pair one column with a single expression.

Single-Column Format

> " *column* = { *expression* | (*singleton_select*) | NULL | *collection_var* } "

Element	Description	Restrictions	Syntax
column	Column to be updated	Cannot be a serial data type	Identifier on page 1231
collection_var	Host or program variable	Must be declared as a collection data type	Language specific
expression	Returns a value for *column*	Cannot contain aggregate functions	Expression on page 1003
singleton_select	Subquery that returns exactly one row	Returned subquery values must have a 1-to-1 correspondence with *column* list	SELECT statement on page 690

You can use this syntax to update a column that has a ROW data type.

You can include any number of "single *column* = single *expression*" terms. The *expression* can be an SQL subquery (enclosed between parentheses) that returns a single row, provided that the corresponding *column* is of a data type that can store the value (or the set of values) from the row that the subquery returns.

To specify values of a ROW-type column in a SET clause, see Updating ROW-Type Columns on page 876. The following examples illustrate the single-column format of the SET clause.

```
UPDATE customer
   SET address1 = '1111 Alder Court', city = 'Palo Alto',
      zipcode = '94301' WHERE customer_num = 103;

UPDATE stock
   SET unit_price = unit_price * 1.07;
```

Updating a Column to NULL

Use the NULL keyword to modify a column value when you use the UPDATE statement. For example, for a customer whose previous address required two address lines but now requires only one, you would use the following entry:

```
UPDATE customer
   SET address1 = '123 New Street',
   SET address2 = null,
   city = 'Palo Alto',
   zipcode = '94303'
   WHERE customer_num = 134;
```

Updating the Same Column Twice

You can specify the same column more than once in the SET clause. If you do so, the column is set to the last value that you specified for the column. In the next example, the **fname** column appears twice in the SET clause. For the row where the customer number is `101`, the user sets **fname** first to `gary` and then to `harry`. After the UPDATE statement executes, the value of **fname** is `harry`.

```
UPDATE customer
   SET fname = "gary", fname = "harry"
      WHERE customer_num = 101;
```

Multiple-Column Format

Use the multiple-column format of the SET clause to list multiple columns and set them equal to corresponding expressions.

(explicit id unique_1262_Connect_42_mucf003) unique_1262_Connect_42_mucf003

Multiple-Column Format

" { (*column*) | * } "

" = "

" ({ | *expression* | { (*singleton_select*) | NULL } }) "

Element	Description	Restrictions	Syntax
column	Name of a column to be updated	Cannot have a serial or ROW type. The number of *column* names must equal the number of values returned to the right of the = sign.	Identifier on page 1231
expression	Expression that returns a value for a *column*	Cannot include aggregate functions	Expression on page 1003
singleton_ select	Subquery that returns exactly one row	Values that the subquery returns must correspond to columns in the *column* list	SELECT statement on page 690
SPL function	SPL routine that returns one or more values	Returned values must have a 1-to-1 correspondence to columns in *the column* list	Identifier on page 1231

The multiple-column format of the SET clause offers the following options for listing a set of columns that you intend to update:

- Explicitly list each column, placing commas between columns and enclosing the set of columns between parentheses.
- Implicitly list all columns in the table by using an asterisk (*).

You must list each expression explicitly, placing comma (,) separators between expressions and enclosing the set of expressions between parentheses. The number of columns must equal the number of values returned by the expression list, unless the expression list includes an SQL subquery.

The following examples show the multiple-column format of the SET clause:

```
UPDATE customer
   SET (fname, lname) = ('John', 'Doe') WHERE customer_num = 101;

UPDATE manufact
   SET * = ('HNT', 'Hunter') WHERE manu_code = 'ANZ';
```

Updating ROW-Type Columns

Use the SET clause to update a named or unnamed ROW-type column. For example, suppose you define the following named ROW type and a table that contains columns of both named and unnamed ROW types:

```
CREATE ROW TYPE address_t
(
   street CHAR(20), city CHAR(15), state CHAR(2)
);
CREATE TABLE empinfo
(
   emp_id INT
   name ROW ( fname CHAR(20), lname CHAR(20)),
   address address_t
);
```

To update an unnamed ROW type, specify the ROW constructor before the parenthesized list of field values.

The following statement updates the **name** column (an unnamed ROW type) of the **empinfo** table:

```
UPDATE empinfo SET name = ROW('John','Williams') WHERE emp_id =455;
```

To update a named ROW type, specify the ROW constructor before the list (in parentheses) of field values, and use the cast (**::**) operator to cast the ROW value as a named ROW type. The following statement updates the **address** column (a named ROW type) of the **empinfo** table:

```
UPDATE empinfo
   SET address = ROW('103 Baker St','Tracy','CA')::address_t
   WHERE emp_id = 3568;
```

For more information on the syntax for ROW constructors, see Constructor Expressions on page 1049. See also Literal Row on page 1199.

The ROW-column SET clause can only support literal values for fields. To use an ESQL/C variable to specify a field value, you must select the ROW data into a **row** variable, use host variables for the individual field values, then update the ROW column with the **row** variable. For more information, see Updating a Row Variable (ESQL/C) on page 882.

You can use host variables to insert *non-literal* values as:

- An entire row type into a column

  Use a **row** variable as a variable name in the SET clause to update all fields in a ROW column at one time.
- Individual fields of a ROW type

  To insert non-literal values into a ROW-type column, you can first update the elements in a **row** variable and then specify the **collection** variable in the SET clause of an UPDATE statement.

When you use a **row** variable in the SET clause, the **row** variable must contain values for each field value. For information on how to insert values into a **row** variable, see Updating a Row Variable (ESQL/C) on page 882.

You can use the UPDATE statement to modify only some of the fields in a row:

- Specify the field names with field projection for all fields whose values remain unchanged.

  For example, the following UPDATE statement changes only the **street** and **city** fields of the **address** column of the **empinfo** table:

  ```
  UPDATE empinfo
     SET address = ROW('23 Elm St', 'Sacramento', address.state)
     WHERE emp_id = 433;
  ```

  The **address.state** field remains unchanged.

- Select the row into an ESQL/C **row** variable and update the desired fields.

  For more information, see Updating a Row Variable (ESQL/C) on page 882.

Updating Values in Opaque-Type Columns

Some opaque data types require special processing when they are updated. For example, if an opaque data type contains spatial or multirepresentational data, it might provide a choice of how to store the data: inside the internal structure or, for large objects, in a smart large object.

This processing is accomplished by calling a user-defined support function called **assign( )**. When you execute UPDATE on a table whose rows contain one of these opaque types, the database server automatically invokes the **assign( )** function for the type. This function can make the decision of how to store the data. For more information about the **assign( )** support function, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

Data Types in Distributed UPDATE Operations

Distributed UPDATE operations across tables in databases of different server instances can return only a subset of the data types that distributed UPDATE operations can return from tables that are all in databases of the same Informix® instance.

The UPDATE statement (or any other SQL data-manipulation language statement) that accesses a database of another Informix® instance can reference only the following data types:

- Built-in data types that are not opaque or complex
- BOOLEAN
- BSON
- JSON
- LVARCHAR
- DISTINCT of built-in types that are not opaque
- DISTINCT of BOOLEAN
- DISTINCT of BSON
- DISTINCT of JSON
- DISTINCT of LVARCHAR
- DISTINCT of the DISTINCT types in this list.

Cross-server distributed UPDATE operations can support these DISTINCT types only if the DISTINCT types are cast explicitly to built-in types, and all of the DISTINCT types, their data type hierarchies, and their casts are defined exactly the same way in each participating database. For additional information about the data types that Informix® supports in cross-server DML operations, see Data Types in Cross-Server Transactions on page 702.

Cross-database distributed UPDATE operations that access other databases of the local Informix® instance, however, can access the cross-server data types in the preceding list, and also the following data types:

- Most of the *built-in opaque data types*, as listed in Data Types in Cross-Database Transactions on page 700
- DISTINCT of the built-in types that are referenced in the preceding line
- DISTINCT of any of the data types that are listed in either of the two preceding lines
- Opaque user-defined data types (UDTs) that are explicitly cast to built-in data types.

Cross-database UPDATE operations can support these DISTINCT types and opaque UDTs only if all the opaque UDTs and DISTINCT types are cast explicitly to built-in types, and all of the opaque UDTs, DISTINCT types, data type hierarchies, and casts are defined exactly the same way in each of the participating databases.

Distributed UPDATE transactions cannot access the database of another Informix® instance unless both servers define TCP/IP or IPCSTR connections in their DBSERVERNAME or DBSERVERALIASES configuration parameters and in the **sqlhosts** file or SQLHOSTS registry subkey. The requirement, that both participating servers support the same type of connection (either TCP/IP or else IPCSTR), applies to any communication between Informix® instances, even if both reside on the same computer.

WHERE Clause of UPDATE

The WHERE clause lets you specify search criteria to limit the rows to be updated. If you omit the WHERE clause, every row in the table is updated. For more information, see the WHERE clause of SELECT on page 731.

SQLSTATE Values When Updating an ANSI-Compliant Database

If you update a table in an ANSI-compliant database with an UPDATE statement that contains the WHERE clause and no rows are found, the database server issues a warning.

You can detect this warning condition in either of the following ways:

- The GET DIAGNOSTICS statement sets the **RETURNED_SQLSTATE** field to the value `02000`. In an SQL API application, the **SQLSTATE** variable contains this same value.
- In an SQL API application, the **sqlca.sqlcode** and **SQLCODE** variables contain the value `100`.

The database server also sets **SQLSTATE** and **SQLCODE** to these values if the UPDATE ... WHERE statement is part of a multistatement PREPARE and the database server returns no rows.

SQLSTATE Values When Updating a Non-ANSI Database

In a database that is not ANSI compliant, the database server does not return a warning when it finds no matching rows for the WHERE clause of an UPDATE statement. The **SQLSTATE** code is `00000` and the **SQLCODE** code is zero (`0`). If the UPDATE ... WHERE statement is part of a multistatement PREPARE, however, and no rows are returned, the database server issues a warning, and sets **SQLSTATE** to `02000` and sets **SQLCODE** to `100`.

Client-server communication protocols of Informix®, such as SQLI and DRDA®, support **SQLSTATE** code values. For a list of these codes, and for information about how to get the message text, see Using the SQLSTATE Error Status Code on page 526.

Subqueries in the WHERE Clause of UPDATE

The FROM clause of a subquery in the WHERE clause of the UPDATE statement can specify as a data source the same table or view that the Table Options clause of the UPDATE statement specifies. UPDATE operations with subqueries that reference the same table object are supported only if all of the following conditions are true:

- The subquery either returns a single row, or else has no correlated column references.
- The subquery is in the UPDATE statement WHERE clause, using Condition with Subquery syntax.
- No SPL routine in the subquery can reference the same table that UPDATE is modifying.

Unless all of these conditions are satisfied, UPDATE statements that include subqueries that reference the same table or view that the UPDATE statement modifies return error -360.

The following example updates the **stock** table by reducing the **unit_price** value by 5% for a subset of prices. The WHERE clause specifies which prices to reduce by applying the IN operator to the rows returned by a subquery that selects only the rows of the **stock** table where the **unit_price** value is greater than 50:

```
UPDATE stock SET unit_price = unit_price * 0.95
   WHERE unit_price IN
      (SELECT unit_price FROM stock WHERE unit_price > 50);
```

This subquery includes only uncorrelated column references, because its only referenced column is in a table specified in its FROM clause. The requirements listed above are in effect, because the data source of the subquery is the same **stock** table that the Table Options clause of the outer UPDATE statement specifies.

The previous example produces the same results as issuing two separate DML statements:

- The SELECT statement, to return a temporary table, **tmp1**, that contains the same rows from the **stock** table that the subquery returned.
- The UPDATE statement, to issue a subquery of the temporary table as a predicate in its WHERE clause to modify every row of the **stock** table where the **unit_price** matches a value in the temporary table:

```
SELECT unit_price FROM stock WHERE unit_price > 50 INTO TEMP tmp1;
UPDATE stock SET unit_price = unit_price * 0.95
   WHERE unit_price IN ( SELECT * FROM tmp1 );
```

Here is an example of a more complex UPDATE statement that includes multiple uncorrelated subqueries in its WHERE clause:

```
UPDATE t1 SET a = a + 10
   WHERE a > ALL (SELECT a FROM t1  WHERE a > 1) AND
         a > ANY (SELECT a FROM t1  WHERE a > 10) AND
          EXISTS (SELECT a FROM t1  WHERE a > 5);;
```

If an enabled Select trigger is defined on a table that is the data source of a subquery in the WHERE clause of an UPDATE statement that modifies the same table, executing that subquery within the UPDATE statement does not activate the Select trigger. Consider the following program fragment:

```
CREATE TRIGGER selt11 SELECT ON t1 BEFORE
  (UPDATE d1
     SET (c1, c2, c3, c4, c5) =
         (c1 + 1, c2 + 1, c3 + 1, c4 + 1, c5 + 1));

UPDATE t2 SET c1 = c1 +1
   WHERE c1 IN
      (SELECT t1.c1 from t1 WHERE t1.c1 > 10 );
```

In the example above, trigger **selt11** is not activated as part of the UPDATE operation on table **t2**.

A subquery in the WHERE clause of the UPDATE statement can include the UNION or the UNION ALL operator, as in the following example.

```
UPDATE t1 SET a = a + 10 WHERE a in (SELECT a FROM t1 WHERE a > 1
   UNION SELECT a FROM t1, t2 WHERE a < b);
```

If the table that the outer UPDATE statement modifies a typed table within a table hierarchy, Informix® supports all of the following operations that use valid subqueries in the WHERE clause of UPDATE:

- UPDATE on target parent table with subquery (SELECT from parent table)
- UPDATE on target parent table with subquery (SELECT from child table)
- UPDATE on target child table with subquery (SELECT from parent table)
- UPDATE on target child table with subquery (SELECT from child table).

The following program fragment illustrates UPDATE operations with subqueries on typed tables:

```
CREATE ROW TYPE r1 (c1 INT, c2 INT);
CREATE ROW TYPE r2 UNDER r1;
CREATE TABLE t1 OF TYPE r1; -- parent table
CREATE TABLE t2 OF TYPE r2 UNDER t1; -- child  table

UPDATE t1 SET c1 = c1 + 1 WHERE c1 IN
   ( SELECT t1.c1 FROM t1 WHERE t1.c1 > 10);

UPDATE t1 SET c1 = c1 + 1 WHERE c1 IN
   ( SELECT t2.c1 FROM t2 WHERE t2.c1 > 10);

UPDATE t2 SET c1 = c1 + 1 WHERE c1 IN
   ( SELECT t2.c1 FROM t2 WHERE t2.c1 > 10);
```

```
UPDATE t2 SET c1 = c1 + 1 WHERE c1 IN
   ( SELECT t1.c1 FROM t1 WHERE t1.c1 > 10);
```

See the Condition with Subquery on page 979 topic for more information about how to use subqueries that return multiple rows as predicates in the WHERE clause of the UPDATE statement.

Using the WHERE CURRENT OF Clause (ESQL/C, SPL)

Use the WHERE CURRENT OF clause to update the current row of a cursor that was declared FOR UPDATE, or to update the current element of a Collection cursor.

Here the *cursor* name cannot be specified as a host variable.

The current row is the most recently fetched row. Because the UPDATE statement does not advance the cursor to the next row, the current row position within the active set of the cursor is not changed by this operation.

For table hierarchies of Informix®, you cannot use this clause if you are selecting from only one table in a table hierarchy. That is, you cannot use this option if you use the ONLY keyword.

In ESQL/C routines, to include the WHERE CURRENT OF keywords, you must have previously used the DECLARE statement to define the *cursor* with the FOR UPDATE option. If the DECLARE statement that created the cursor specified one or more columns in the FOR UPDATE clause, you are restricted to updating only those columns in a subsequent UPDATE ... WHERE CURRENT OF statement. The advantage to specifying columns in the FOR UPDATE clause of a DECLARE statement is speed. The database server can usually perform updates more quickly if columns are specified in the DECLARE statement.

In SPL routines, you can specify a cursor after the WHERE CURRENT OF keywords in an UPDATE statement only if you declared the *cursor_id* in the FOREACH statement of SPL. You cannot use the DECLARE statement in an SPL routine to declare the name of a dynamic cursor and to associate that cursor with the statement identifier of a prepared object that the PREPARE statement has declared in the same SPL routine.

Note: An Update cursor can perform updates that are not possible with the UPDATE statement.

The following example illustrates the CURRENT OF form of the WHERE clause. In this example, updates are performed on a range of customers who receive 10-percent discounts (assume that a new column, **discount**, is added to the **customer** table). The UPDATE statement is prepared outside the WHILE loop to ensure that parsing is done only once.

```
char answer [1] = 'y';
EXEC SQL BEGIN DECLARE SECTION;
   char fname[32],lname[32];
   int low,high;
EXEC SQL END DECLARE SECTION;
main()
{
   EXEC SQL connect to 'stores_demo';
   EXEC SQL prepare sel_stmt from
      'select fname, lname from customer
       where cust_num between ? and ? for update';
```

```
EXEC SQL declare x cursor for sel_stmt;
   printf("\nEnter lower limit customer number: ");
```

```
   scanf("%d", &low);
   printf("\nEnter upper limit customer number: ");
   scanf("%d", &high);
   EXEC SQL open x using :low, :high;
   EXEC SQL prepare u from
      'update customer set discount = 0.1  where current of x';
   while (1)
      {
      EXEC SQL fetch x into :fname, :lname;
       if ( SQLCODE == SQLNOTFOUND) break;
       }
   printf("\nUpdate %.10s %.10s (y/n)?", fname, lname);
   if (answer = getch() == 'y')
      EXEC SQL execute u;
   EXEC SQL close x;
}
```

Updating a Row Variable (ESQL/C)

About this task

The UPDATE statement with the Collection-Derived Table segment allows you to update fields in a **row** variable. The Collection-Derived Table segment identifies the **row** variable in which to update the fields. For more information, see Collection-Derived Table on page 1212.

To update fields

1. Create a **row** variable in your program.
2. Optionally, select a ROW-type column into the **row** variable with the SELECT statement (without the Collection-Derived Table segment).
3. Update fields of the **row** variable with the UPDATE statement and the Collection-Derived Table segment.
4. After the **row** variable contains the correct fields, you then use the UPDATE or INSERT statement on a table or view name to save the **row** variable in the ROW column (named or unnamed).

Results

The UPDATE statement and the Collection-Derived Table segment allow you to update a field or a group of fields in the **row** variable. Specify the new field values in the SET clause. For example, the following UPDATE changes the **x** and **y** fields in the **myrect row** variable:

```
EXEC SQL BEGIN DECLARE SECTION;
   row (x int, y int, length float, width float) myrect;
EXEC SQL END DECLARE SECTION;
. . .
EXEC SQL select into :myrect from rectangles where area = 64;
EXEC SQL update table(:myrect) set x=3, y=4;
```

Suppose that after the SELECT statement, the **myrect2** variable has the values `x=0`, `y=0`, `length=8`, and `width=8`. After the UPDATE statement, the **myrect2** variable has field values of `x=3`, `y=4`, `length=8`, and `width=8`. You cannot use a **row** variable in the Collection-Derived Table segment of an INSERT statement.

You can, however, use the UPDATE statement and the Collection-Derived Table segment to insert new field values into a **row** host variable, if you specify a value for every field in the row.

For example, the following code fragment inserts new field values into the **row** variable **myrect** and then inserts this **row** variable into the database:

```
EXEC SQL update table(:myrect)
   set x=3, y=4, length=12, width=6;
EXEC SQL insert into rectangles
   values (72, :myrect);
```

If the **row** variable is an untyped variable, you must use a SELECT statement *before* the UPDATE so that can determine the data types of the fields. An UPDATE of fields in a **row** variable cannot include a WHERE clause.

The **row** variable can store the field values of the row, but it has no intrinsic connection with a database column. Once the **row** variable contains the correct field values, you must then save the variable into the ROW column with one of the following SQL statements:

- To update the ROW column in the table with contents of the **row** variable, use an UPDATE statement on a table or view name and specify the **row** variable in the SET clause. (For more information, see Updating ROW-Type Columns on page 876.)
- To insert a **row** into a column, use the INSERT statement on a table or view name and specify the **row** variable in the VALUES clause. (For more information, see Inserting Values into ROW-Type Columns on page 586.)

For examples of SPL ROW variables, see the *HCL® Informix® Guide to SQL: Tutorial*. For more information on using **row** variables, see the discussion of complex data types in the *HCL® Informix® Enterprise Replication Guide*.

UPDATE STATISTICS statement

Use the UPDATE STATISTICS statement to update system catalog information that the query optimizer uses for operations on objects in the local database. The UPDATE STATISTICS statement is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_383_Connect_42_upsta003) unique_383_Connect_42_upsta003(explicit id unique_383_Connect_42_upsta004)

unique_383_Connect_42_upsta004 **UPDATE STATISTICS** { { { **LOW** <Table and Column Scope> [**DROP DISTRIBUTIONS** [**ONLY**]] | { **MEDIUM** | **HIGH** } <Table and Column Scope> [<Resolution Clause>] } [{ **FORCE** | **AUTO** }] } | <Routine Statistics> }

Table and Column Scope

" [**FOR TABLE** { [[*'owner'* .] { *table* | *synonym* } [(*column*)]] | [**ONLY** ([*'owner'* .] { *table* | *synonym* }) [(*column*)]] }] "

Element	Description	Restrictions	Syntax
column	A column in table or synonym	Must exist. With the MEDIUM or HIGH keyword, the column cannot be of BYTE, LVARCHAR, or TEXT data type.	Identifier on page 1231
owner	The owner of table or synonym	Must be the owner of table or synonym	Owner name on page 1262
synonym	A synonym for a table whose statistics are to be updated	The synonym and the table to which it points must exist in the current database	Identifier on page 1231
table	Table for which statistics are to be updated	Must exist in the current database or be a temporary table created in the current session	Identifier on page 1231

Usage

Use the UPDATE STATISTICS statement to perform any of the following tasks:

- Calculate the distribution of column values for tables and table fragments.
- Update system catalog tables that the database server uses to optimize queries.
- Force reoptimization of SPL routines.
- Convert existing indexes when you upgrade the database server.

Run the UPDATE STATISTICS statement in a transaction that does not contain any other statements.

The Table and Column Scope clause and the Routine Statistics clause can be empty. If you specify no mode, no table, no routine, and no Resolution clause, the default scope of the UPDATE STATISTICS statement is all the permanent tables in the current database, as in the following example.

```
UPDATE STATISTICS;  -- calculates LOW column distribution statistics for
                    -- all permanent user-defined and system catalog tables
                    -- that the AUTO_STAT_MODE and STATCHANGE settings allow
```

For more examples of UPDATE STATISTICS statements that specify no identifiers of tables or of SPL routines, see the topic The scope of UPDATE STATISTICS statements on page 885. See also the references to AUTO_STAT_MODE and STATCHANGE topics in Performance considerations of UPDATE STATISTICS statements on page 902.

The UPDATE STATISTICS statement is not supported on secondary servers within a high-availability cluster.

Restriction: You cannot update the statistics for a table or update the query plan of a UDR in any database except the current database. That is, the database server ignores remote database objects when executing the UPDATE STATISTICS statement.

Important: The statistics that the database server collects might require an sbspace for storage. You can create an sbspace by running the onspaces -c -S command. You must also set the SYSSBSPACENAME configuration parameter to the sbspace name. If the SYSSBSPACENAME configuration parameter is not set, the database server

might not be able to store the column distribution statistics, so that the UPDATE STATISTICS statement fails with error -9814, "`Invalid default sbspace name`".

The scope of UPDATE STATISTICS statements

The scope of UPDATE STATISTICS can be restricted by whatever identifiers of tables or columns follow the `FOR TABLE` keywords, or by whatever identifiers of SPL routines follow the `FOR FUNCTION`, `FOR PROCEDURE`, or `FOR ROUTINE` keywords.

Scope of UPDATE STATISTICS for tables

The specifications that follow the STATISTICS keyword determine the scope of UPDATE STATISTICS statements.

- If the UPDATE STATISTICS statement includes no `FOR` keyword, then the scope cannot be explicitly restricted to a single table or to a single SPL routine.

  In the case of no syntax tokens following the `STATISTICS` keyword, the default scope is to update statistics in `LOW` mode for every permanent table in the current database, including its system catalog tables, and to update the statistics in the **systables**, **syscolumns**, and **sysindices** system catalog tables.

  Both of the following examples produce the same result:

  ```
  UPDATE STATISTICS;
  UPDATE STATISTICS LOW;
  ```

  For database tables with distributed storage, the CREATE TABLE or ALTER TABLE statement that defined the Statistics Options properties of the table determines the table-level or fragment-level granularity of these distribution statistics.

  If you include only the `MEDIUM` or `HIGH` keyword, the scope is the same, although the resources required for the operation are greater.

- If you include only the `FOR TABLE` keywords without also specifying the name or synonym of a table, the database server recalculates distributions on all of the tables in the current database, and on all of the temporary tables in your session. UPDATE STATISTICS has no effect, however, on objects defined by the CREATE EXTERNAL TABLE statement.
- If you specify a table after the `FOR TABLE` keywords without also specifying a list of columns, the database server recalculates the statistical distributions on all of the columns of the specified table.
- If you specify a table after the `FOR TABLE` keywords and a list of columns, the database server recalculates the statistical distributions of only the specified columns.

In summary, for UPDATE STATISTICS statements that include no database object names, the database server updates distribution statistics on the following table objects by default:

- If you specify no scope or mode,

  ```
  UPDATE STATISTICS;
  ```

  distribution statistics are calculated in `LOW` mode for all permanent tables in the database.
- If you specify the scope as `FOR TABLE`, but with no list of tables,

```
UPDATE STATISTICS MEDIUM FOR TABLE;
```

distribution statistics are calculated in the specified mode for all permanent tables in the database and for all temporary tables in the session. In the example above, `MEDIUM` specifies a more costly statistics mode than the default LOW mode, but has no effect on which table objects are in scope.

The scope of UPDATE STATISTICS for SPL routines

For UPDATE STATISTICS statements that include the `FOR FUNCTION`, `FOR PROCEDURE`, or `FOR ROUTINE` keywords, the scope depends on whether you also include the name of a routine and an argument list.

- If you specify the scope as FOR PROCEDURE, or FOR ROUTINE, or FOR FUNCTION, but with no list of SPL routines, as in these examples,

  ```
  UPDATE STATISTICS FOR FUNCTION;
  UPDATE STATISTICS FOR PROCEDURE;
  UPDATE STATISTICS FOR ROUTINE;
  ```

  the database server takes these actions:
  - reoptimizes the execution plans for the DML statements in every SPL routine in the database,
  - and updates the **sysprocplan** system catalog table with the reoptimized execution plan.

  It does not, however, refresh any table distribution statistics.

The scope for overloaded SPL routines

An UPDATE STATISTICS statement that specifies the identifier of an overloaded SPL routine identifier, where more than one routine have the same name, but different parameters, can reoptimize more than one SPL routine.

- If you specify the scope with the name of an SPL routine, but with no list of parameters,

  ```
  UPDATE STATISTICS FOR FUNCTION someSPLroutine;
  UPDATE STATISTICS FOR PROCEDURE someSPLroutine;
  UPDATE STATISTICS FOR ROUTINE someSPLroutine;
  ```

  the execution plans are reoptimized for the DML statements in any SPL routine with the specified identifier in the database. If the routine name is overloaded, because more than one SPL routine is registered in the database with that name, every SPL routine with that name is reoptimized.
- If you specify the scope with the name of an overloaded SPL routine and a parameter list,

  ```
  UPDATE STATISTICS FOR FUNCTION someSPLroutine(INT, CHAR(140));
  UPDATE STATISTICS FOR PROCEDURE someSPLroutine(INT, CHAR(140));
  UPDATE STATISTICS FOR ROUTINE someSPLroutine(INT, CHAR(140));
  ```

  the execution plans are reoptimized for the DML statements in any SPL routine with the specified identifier and the same argument types in the database.

If the name of an SPL routine is not overloaded, however, you can omit the parameter list and use the SPECIFIC keyword to restrict the scope of the UPDATE STATISTICS statement to the specified SPL routine:

```
UPDATE STATISTICS FOR SPECIFIC FUNCTION someSPLroutine;
UPDATE STATISTICS FOR SPECIFIC PROCEDURE someSPLroutine;
UPDATE STATISTICS FOR SPECIFIC ROUTINE someSPLroutine;
```

No other routine will be reoptimized, if the specified name is unique among SPL routine names in the database.

Restricting the scope for tables with automatic mode

When the automatic mode of UPDATE STATISTICS is enabled for the database or for only the current session, the default scope of UPDATE STATISTICS can be further reduced to tables and table fragments with stale distribution statistics. Only these are refreshed, if the percentage of rows that subsequent DML operations have changed exceeds a user-defined STATCHANGE threshold. This restriction can apply to the database, to the session, or to individual tables, if you set their thresholds in the Statistics Options clause of CREATE TABLE or ALTER TABLE statements.

For example, if the AUTO_STAT_MODE setting is ON for the session, and one or more tables that the Perform_work function processed have a high STATCHANGE threshold, you might decide to override the selective refreshing of table statistics by including the `FORCE` keyword:

```
UPDATE STATISTICS FOR SPECIFIC FUNCTION Perform_work FORCE;
```

This overrides the restriction to tables with stale statistics, so that the execution plan for Perform_work will be based on the current data values, whether or not the column distribution statistics in the system catalog satisfy the STATCHANGE criteria for stale data,

For more information about automating the selective recalculation of distribution statistics, see the #unique_1274 and AUTO_STAT_MODE session environment option topics. For information about the performance advantages of restricting the scope of UPDATE STATISTICS statements, see Performance considerations of UPDATE STATISTICS statements on page 902.

Updating Statistics for Tables

Although a change to the database might make information in the **systables**, **syscolumns**, **sysindices**, **sysfragments**, **sysdistrib**, and **sysfragdist** system catalog tables obsolete, the database server does not automatically update those tables after most SQL statements.

Issue an appropriate UPDATE STATISTICS statement in the following situations to ensure that the column distribution information in the system catalog tables reflects the current state of the database:

- You perform extensive modifications to a table.
- An application changes the distribution of column values.

  The UPDATE STATISTICS statement refreshes the data distribution statistics that the database server uses to optimize queries on the modified objects.
- You upgrade a database for use with a newer database server.

  The UPDATE STATISTICS statement converts the old indexes to conform to the newer database server index format and implicitly drops the old indexes.

  You can convert the indexes table by table or for the entire database at one time. Follow the conversion guidelines in the *HCL Informix® Migration Guide*.

If your application makes many modifications to the data in a particular table, update the system catalog for that table routinely with UPDATE STATISTICS to improve query efficiency. The term *many modifications* is relative to the resolution of the distributions. If the data modifications have little effect on the distribution of column values, you do not need to execute UPDATE STATISTICS.

Distribution Statistics in NLSCASE INSENSITIVE databases

In a database created with the NLSCASE INSENSITIVE property, database server operations on columns and expressions of NCHAR or NVARCHAR data types make no distinction between upper case and lower case letters. In data sets that include strings of the same sequence of letters but with case variants, generating the data distributions of NCHAR and NVARCHAR columns requires fewer bins than in a case-sensitive database containing the same records. The database server identifies all the case-variant values as only a single distinct value, and uses this result when it generates the column, index, or fragment-level statistics.

For more information about NLSCASE INSENSITIVE databases, see Duplicate rows in NLSCASE INSENSITIVE databases on page 699, Specifying NLSCASE case sensitivity on page 195, and NCHAR and NVARCHAR expressions in case-insensitive databases.

Automated Table Statistics Maintenance

To simplify the complex and repetitive task of the DBA in maintaining current table statistics from which the query plan optimizer can design efficient query plans, Informix® provides a table statistics maintenance system, called Auto Update Statistics (AUS). This can automate the identification of tables whose statistics are stale, and can automate the construction and execution of the corresponding UPDATE STATISTICS statements to recalculate their column distributions. The AUS system is provided with built-in criteria for when table statistics should be updated, but the DBA can modify these criteria to reflect current requirements and workloads.

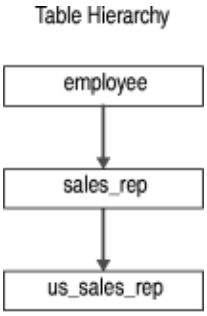
For more information, see the description of the Auto Update Statistics maintenance system in the *HCL® Informix® Performance Guide*. See also the description of the Scheduler, which the DBA can use to specify the policies, resources, and frequency with which the AUS system recalculates table statistics, in the *HCL® Informix® Administrator's Guide*.

Using the FOR TABLE ONLY Keywords

Use the FOR TABLE ONLY keywords to collect data for a single table within a hierarchy of typed tables. If you do not include the ONLY keyword immediately after FOR TABLE when the specified *table* has subtables, Informix® creates distributions for that table and also for every subtable under it in the hierarchy.

For example, suppose your database has the typed table hierarchy that appears in Figure 10: Example of Typed Table Hierarchy on page 889, which shows a supertable named **employee** that has a subtable named **sales_rep**. The **sales_rep** table, in turn, has a subtable named **us_sales_rep.**

Figure 10. Example of Typed Table Hierarchy



When the following statement executes, the database server generates statistics on both the **sales_rep** and **us_sales_rep** tables:

```
UPDATE STATISTICS FOR TABLE sales_rep;
```

In contrast, the following example generates statistical data for each column in table **sales_rep** but does not act on tables **employee** or **us_sales_rep**:

```
UPDATE STATISTICS FOR TABLE ONLY (sales_rep);
```

If you specify FOR TABLE ONLY, as in this example, the identifier of the table (or *owner*.*table*) must be enclosed between parentheses.

Because neither of the previous examples specified the level at which to update the statistical data, the database server uses the LOW mode by default.

Updating Statistics for Columns

The Table and Column Scope specification can also include the names of one or more columns for which you want statistical distributions calculated.

In the following example, this statement calculates the distributions for three columns of the **orders** table:

```
UPDATE STATISTICS FOR TABLE orders (order_num, customer_num, ship_date);
```

If you include no *column* name in the FOR TABLE clause, then distributions are calculated for all columns of the specified *table*, using the LOW, MEDIUM, or HIGH mode and the number of bins implied by the specified or default Resolution clause *percentage* (for MEDIUM or HIGH mode) that you request.

See also Updating Statistics for Columns of User-Defined Types on page 890 for UPDATE STATISTICS restrictions on columns that store UDTs.

Examining Index Pages

In Informix®, when you execute the UPDATE STATISTICS statement in any mode, the database server reads through index pages to:

- Compute statistics for the query optimizer
- Locate pages that have the delete flag marked as `1`

If pages are found with the delete flag marked as `1`, the corresponding keys are removed from the B-tree cleaner list.

This operation is particularly useful if a system failure causes the B-tree cleaner list (which exists in shared memory) to be lost. To remove the B-tree items that have been marked as deleted but are not yet removed from the B-tree, run the UPDATE STATISTICS statement. For information on the B-tree cleaner list, see your *HCL® Informix® Administrator's Guide*.

Updating Statistics for Columns of User-Defined Types

To calculate statistics for a column of a user-defined data type (UDT), you must use either the UPDATE STATISTICS MEDIUM FOR TABLE or the UPDATE STATISTICS HIGH FOR TABLE statement.

Without the `MEDIUM` or `HIGH` keyword, the UPDATE STATISTICS FOR TABLE statement ignores UDT columns.

Restrictions on UDT statistics

The UPDATE STATISTICS statement does not collect values for the **colmin** and **colmax** columns of the **syscolumns** system catalog table for columns that hold user-defined data types.

To drop distribution statistics for columns that store values of user-defined data types, you must execute UPDATE STATISTICS in LOW mode and include the `DROP DISTRIBUTIONS` keywords.

When you run the UPDATE STATISTICS LOW FOR TABLE DROP DISTRIBUTIONS statement, the database server deletes the row in the **sysdistrib** system catalog table that corresponds to the **tableid** and **colno** values for the column. In addition, the database server removes any large objects that might have been created to store distribution statistics for the specified opaque column.

Requirements for Statistics on Opaque Columns

UPDATE STATISTICS can collect statistics for columns of user-defined opaque data types only if support routines for **statcollect( )**, **statprint( )**, and the selectivity functions are defined for the UDTs. You must also hold Usage privilege on these routines.

In some cases, UPDATE STATISTICS also requires an sbspace as specified by the SYSSBSPACENAME configuration parameter. For information about how to provide statistical data for a column whose data type is a UDT, refer to the *HCL® Informix® DataBlade® API Programmer's Guide*. For information about SYSSBSPACENAME, refer to your *HCL® Informix® Administrator's Reference*.

Using the FORCE and AUTO keywords

In sessions or in databases where the automatic UPDATE STATISTICS mode is enabled, you can optionally use either the `FORCE` keyword or the `AUTO` keyword to control the scope of the UPDATE STATISTICS statement when it updates the current distribution statistics of tables and columns in the system catalog. These keywords affect only table and fragment statistics, and are not valid in operations on SPL routine statistics.

If you omit both the `FORCE` keyword and the `AUTO` keyword, the effect of the UPDATE STATISTICS statement on table and fragment distribution statistics is determined by the explicit or default setting of the AUTO_STAT_MODE configuration

parameter, unless the AUTO_STAT_MODE session environment variable is set to override that configuration parameter for the current session.

Specifying either of the `FORCE` or `AUTO` keywords affects only the current UPDATE STATISTICS operation. The database server issues an exception if you attempt to include both the `FORCE` and the `AUTO` keywords in the same UPDATE STATISTICS statement.

The FORCE keyword

The FORCE keyword refreshes the statistics for all tables and columns within the specified scope. If automatic mode for the UPDATE STATISTICS statement is enabled, the `FORCE` keyword overrides automatic mode, so that values of the STATCHANGE attributes of tables and fragments within the scope of the FOR TABLE specification are ignored, as if the AUTO_STAT_MODE setting were OFF for the current UPDATE STATISTICS FORCE operation.

The two examples that follow apply the `FORCE` keyword, respectively, to the default table scope of UPDATE STATISTICS, and to a nonfragmented individual table called **tableN**.

```
UPDATE STATISTICS FORCE;
UPDATE STATISTICS MEDIUM FOR TABLE tableN FORCE;
```

The first statement instructs the database server to take the following actions:

- Recalculate the distribution statistics in LOW mode for every permanent table and table fragment in the database, ignoring whether statistics for each table are stale according to their STATCHANGE thresholds.
- Store the new distributions in the system catalog tables.

The second statement instructs the database server to take the following actions:

- Recalculate the column distribution statistics in MEDIUM mode for table **tableN**.
- Update all the system catalog tables that store distribution statistics for that table.

Including the `FORCE` keyword emulates the previous UPDATE STATISTICS behavior of Informix® database servers before version 11.70.

The AUTO keyword

The AUTO keyword causes the database server to run the UPDATE STATISTICS statement in automatic mode, but only for tables and fragments whose statistics are missing or stale. The distribution statistics are not refreshed for any tables or fragments whose STATCHANGE value is below the specified threshold.

The following statements specify the `AUTO` keyword.

```
UPDATE STATISTICS AUTO;
UPDATE STATISTICS MEDIUM FOR TABLE tableN AUTO;
```

The first statement instructs the database server to take the following actions:

- Examine every permanent table in the database, and recalculate only the missing or stale data distribution statistics for tables and fragments in which the percentage of new, deleted, or changed rows since the table distributions were last calculated exceeds the STATCHANGE threshold for that table. This might be an empty set, if no tables or table fragments exceed their STATCHANGE thresholds.
- If any tables or fragments qualified to be recalculated, store their new statistics in the appropriate system catalog tables.

The second statement instructs the database server to take the following actions: .

- If the current distribution statistics for **tableN** do not exceed the STATCHANGE threshold for that table, take no action.
- If the percentage of new, deleted, or changed rows indicate that the current distributions are stale, recalculate the column distribution statistics for table **tableN** in MEDIUM mode, and update all the system catalog tables that store distribution statistics for that table.

When sufficiently accurate statistics are already available to the query optimizer for some tables or table fragments, the AUTO option avoids unnecessary recalculations. In that case, an UPDATE STATISTICS AUTO operation requires less time, without detriment to query performance, than a corresponding UPDATE STATISTICS FORCE operation.

Using the LOW mode option

Use the LOW option of the UPDATE STATISTICS statement to generate and update some of the relevant statistical data regarding table, row, and page-count statistics in the **systables** system catalog table. If you do not specify any mode, the LOW mode is the default.

In Informix®, the LOW mode also generates and updates some index and column statistics for specified columns in the **syscolumns** and the **sysindexes** system catalog tables.

The LOW mode generates the least amount of information about the column. If you want the UPDATE STATISTICS statement to do minimal work, specify a column that is not part of an index. The **colmax** and **colmin** values in **syscolumns** are not updated unless there is an index on the column.

The following example updates statistics on the **customer_num** column of the **customer** table:

```
UPDATE STATISTICS LOW FOR TABLE customer (customer_num);
```

Because the LOW mode option does not update data in the **sysdistrib** system catalog table, all distributions associated with the **customer** table remain intact, even those that already exist on the **customer_num** column.

You can set the **USTLOW_SAMPLE** configuration parameter or the USTLOW_SAMPLE option of the SET ENVIRONMENT statement to enable sampling during the gathering of index statistics for UPDATE STATISTICS operations in LOW mode, as in the following example:

```
SET ENVIRONMENT USTLOW_SAMPLE ON;
UPDATE STATISTICS LOW FOR TABLE items (quantity,total_price);
```

If the **USTLOW_SAMPLE** configuration parameter value is zero, the default behavior for all sessions of the server instance is for LOW mode sampling to be disabled. In this example, however, the UPDATE STATISTICS LOW statement uses sampling,

because the `ON` setting in the preceding SET ENVIRONMENT USTLOW_SAMPLE statement overrides the **USTLOW_SAMPLE** setting for subsequent LOW mode statistical operations in the same session. For more information, see USTLOW_SAMPLE environment option.

Using the DROP DISTRIBUTIONS Option

Use the DROP DISTRIBUTIONS option to force the removal of distribution information from the **sysdistrib** system catalog table.

When you specify the DROP DISTRIBUTIONS option, the database server removes the existing distribution data for the column or columns that you specify. If you do not specify any columns, the database server removes all the distribution data for that table.

You must have the DBA privilege or be owner of the table to use this option.

The following example shows how to remove distributions for the **customer_num** column in the **customer** table:

```
UPDATE STATISTICS LOW
   FOR TABLE customer (customer_num) DROP DISTRIBUTIONS;
```

As the example shows, you drop the distribution data at the same time you update the statistical data that the LOW mode option generates.

Using the DROP DISTRIBUTIONS ONLY Option

Use the DROP DISTRIBUTIONS ONLY option to remove distribution information from the **sysdistrib** table and update the **systables.version** column in the system catalog for those tables whose distributions were dropped, without gathering any LOW mode table and index statistics.

If you specify both the DROP DISTRIBUTIONS ONLY option and the FOR TABLE clause, Informix® removes the existing distribution data for the set of columns of the *table* that the FOR TABLE clause specifies (or for all columns, if you provide no *column* specification), but does not gather any LOW mode table and index statistics.

You must have the DBA privilege or be owner of the table to use this option.

The following example removes distributions for the **customer_num** column in the **customer** table:

```
UPDATE STATISTICS LOW
   FOR TABLE customer (customer_num) DROP DISTRIBUTIONS ONLY;
```

This drops the **customer.customer_num** distribution data without updating the statistical information that the LOW mode option generates when the ONLY keyword does not follow the DROP DISTRIBUTIONS keywords. This example deletes from the system catalog any row describing **customer.customer_num** from the **sysdistrib** table, and updates the **version** number for **customer** in the **systables** table. None of the other LOW mode updates are performed on **systables**, so the **nrow** and **npused** column values are unchanged by this example, and the **syscolumns**, **sysfragments** and **sysindexes** tables of the system catalog are not updated. The LOW keyword has no effect in this example, but the DROP DISTRIBUTIONS ONLY option is not available in MEDIUM or HIGH mode.

Because it specifies no FOR TABLE clause, the next example drops all rows from the **sysdistrib** table and updates the **systables.version** column in the system catalog for all tables in the database.

```
UPDATE STATISTICS DROP DISTRIBUTIONS ONLY;
```

Using the MEDIUM mode option

Use the MEDIUM mode option to update the same statistics that you can perform with the LOW mode option,and also generate statistics about the distribution of data values for each specified column.

After UPDATE STATISTICS MEDIUM has been run on a table, the query optimizer typically chooses a more efficient execution plan, compared to the same SELECT statement when only LOW mode column distribution statistics are available for the table. column

The database server places distribution information in the **sysdistrib** system catalog table, and in other system catalog tables for fragmented tables that use distributed storage.

If you use the MEDIUM mode option, the database server scans tables at least once and takes longer to execute on a given table than the LOW mode option.

When you use the MEDIUM mode option, the data distributions are obtained by sampling a percentage of data rows, using a statistical confidence level that you specify, or else a default confidence level of 95 percent. You can also specify an explicit minimum sampling size in the Resolution clause. Because the MEDIUM sample size is usually much smaller than the actual number of rows, this mode executes more quickly than the HIGH mode.

In distributions obtained by sampling, the results can vary, because different samples of rows can have different sampling errors. If the results vary significantly, you can use the Resolution clause to increase the sampling size, or to lower the *percent*, or to increase the *confidence* level to obtain more consistent results.

If the Resolution clause specifies no *percent* of sampled rows per bin, the default average percentage of the sample in each bin is `2.5`, which divides the range into approximately 40 intervals. If you do not specify a value for *confidence* level, the default level is `0.95`. This value can be roughly interpreted to mean that 95 times out of 100, the difference between the MEDIUM estimate and the exact value from HIGH distributions is not statistically significant.

Distributions are not calculated, however, for LVARCHAR, BYTE, or TEXT columns.

You must have the DBA privilege or be the owner of the table to create MEDIUM mode distributions. For more information on the MEDIUM and HIGH mode options, see the Resolution Clause on page 895.

Using the HIGH mode option

Use the HIGH mode option to update the same statistics that you can calculate with the MEDIUM mode option. The difference between UPDATE STATISTICS HIGH and UPDATE STATISTICS MEDIUM is the number of rows sampled.

UPDATE STATISTICS HIGH scans the entire table, while UPDATE STATISTICS MEDIUM samples only a subset of rows, based on the confidence and resolution used by the UPDATE STATISTICS statement.

For indexed tables that already have MEDIUM mode distribution statistics available for every column, the query optimizer typically chooses more efficient execution plans after you run UPDATE STATISTICS HIGH on every column that is part of an index key.

The database server places distribution information in the **sysdistrib** system catalog table, and in other system catalog tables for fragmented tables that use distributed storage.

If you do not specify a Resolution clause, the default percentage of data distributed to every bin is `0.5`, a value that partitions the range of values for each column into approximately 200 intervals.

The constructed distribution is exact. Because more information is gathered, this mode executes more slowly than LOW or MEDIUM modes. If you use the HIGH mode option of UPDATE STATISTICS, the database server can take considerable time to gather the information across the database, particularly a database with large tables. The HIGH mode might scan each table several times for each column. To minimize processing time, specify a table name and column names within that table, rather than accept the default scope of all tables.

Distributions are not calculated, however, for LVARCHAR, BYTE, or TEXT columns.

You must have the DBA privilege or be the owner of the table to create HIGH mode distributions. For more information on the MEDIUM and HIGH mode options, see the topic Resolution Clause on page 895.

Resolution Clause

Use the Resolution clause in MEDIUM or HIGH mode to adjust the size of the distribution bins, and to avoid calculating data on indexes.

In MEDIUM mode only, you can also use the Resolution clause to specify a lower limit to the sampling size and to adjust the confidence level.

Resolution Clause

" { <Resolution Clause for MEDIUM Mode> <Resolution Clause for HIGH Mode> } "

Resolution Clause for MEDIUM Mode

" [**SAMPLING SIZE** *min*] "

" [**RESOLUTION***percent* [*confidence*]] [**DISTRIBUTIONS ONLY**] "

Resolution Clause for HIGH Mode

" [**RESOLUTION***percent*] "

" [**DISTRIBUTIONS ONLY**] "

Element	Description	Restrictions	Syntax
confidence	Estimated likelihood that sampling in MEDIUM mode produces results equivalent to the exact HIGH mode. Default level is 0.95.	Must be within the range from 0.80 (minimum) to 0.99 (maximum)	Literal Number on page 1198
percent	Average percentage of the sample in each distribution bin. Default is 2.5 for MEDIUM and 0.5 for HIGH.	Minimum value is 1/*nrows*, for *nrows* the number of rows in the table	Literal Number on page 1198
min	The minimum integer number of randomly selected rows on which to generate the data distributions	Must be greater than zero but cannot exceed *nrows*	Literal Number on page 1198

A *distribution* is a mapping of the data in a column into a set of column values, ordered by magnitude or by collation. The range of these sample values is partitioned into disjunct intervals, called *bins*, each containing an approximately equal portion of the sample of column values. For example, if one bin holds 2 percent of the data, approximately 50 such intervals hold the entire sample.

Some statistical texts call these bins *equivalence categories*. Each contains a disjunct subset of the range of the data values that are sampled from the column.

If you include the RESOLUTION keyword, it must be followed by a literal number, specifying the *percent* of values in each bin. In MEDIUM mode, it can be followed by either one or two literal numbers, with the optional second number specifying the *confidence* level, as in this example:

```
UPDATE STATISTICS MEDIUM FOR TABLE orders
   RESOLUTION 4 0.90 DISTRIBUTIONS ONLY;
```

This specifies 4% of the data per bin, implying approximately 25 bins, and a confidence level of 90%, and no examination of index data. If the 0.90 value were omitted, then the default level of confidence would have been in effect. If the RESOLUTION keyword and both numeric values were omitted, then default values for *percent* (2.5%) and for *confidence* (0.95) would be used.

The query optimizer estimates the selectivity of a WHERE clause by examining, for each column included in the WHERE clause, the proportional occurrence of the data values contained in the column.

You cannot create distributions for BYTE or TEXT columns. If you include a BYTE or TEXT column in an UPDATE STATISTICS statement that specifies MEDIUM or HIGH distributions, no distributions are created for those columns. Distributions are constructed for other columns in the list, however, and the statement does not return an error.

Columns of the VARCHAR data type do not use overflow bins, even when multiple bins are being used for duplicate values.

You can use the first two parameters of the **DBUPSPACE** environment variable to constrain the disk space and memory resources that the UPDATE STATISTICS statement can use to sort data when it constructs column distributions. These settings affect performance, because they determine how many times the database server scans the specified table to construct each distribution. (A third **DBUPSPACE** parameter can control whether UPDATE STATISTICS sorts with indexes when calculating column distributions, and whether the **explain** output file stores the plan by which the column distributions are calculated.)

Specifying the SAMPLING SIZE

In MEDIUM mode, you can optionally use the SAMPLING SIZE keywords to specify the minimum number of rows to sample for calculating column distribution statistics. If the Resolution clause omits the RESOLUTION keyword and specifies no *confidence* level and no *percent* value, then the number of rows that Informix® samples will be the larger of the following two values:

- The *min* value that you specify immediately after the SAMPLING SIZE keywords
- The sampling size that is required for the default *percent* of rows in each bin (2.5%) and for the minimum *confidence* level (0.80).

If a sampling size is specified in a Resolution clause that includes explicit values for both the average *percent* of sampled rows per bin and for the *confidence* level, then the number of sampled rows will be the larger of these two values:

- The *min* value that you specify immediately after the SAMPLING SIZE keywords
- The sampling size that is required for the specified *percent* of rows and for the specified *confidence* level.

If a sampling size is specified in a Resolution clause that includes an average *percentage* value but sets no *confidence* level, then the minimum *confidence* value of 0.80 is used to calculate the actual sampling size for Informix® to use if the specified *size* is smaller.

For example, the following statement calculates statistics for two columns of the **customer** table, without updating index information. At least 200 rows will be sampled, but the actual size of the sample might be larger than 200 if more rows are required to provide the default 0.80 confidence level for a sample distribution that uses approximately 50 equivalence categories, with an average percentage of 2% of the sampled values in each bin.

```
UPDATE STATISTICS MEDIUM FOR TABLE customer (city, state)
   SAMPLING SIZE 200 RESOLUTION 2 DISTRIBUTIONS ONLY;
```

Whether or not you include an explicit SAMPLING SIZE specification in the Resolution clause, Informix® records in the system catalog the actual sampling size (as a percentage of the total number of rows in the table) at the time of MEDIUM mode UPDATE STATISTICS creation.

Using the DISTRIBUTIONS ONLY Option to Suppress Index Information

In Informix®, when you specify the DISTRIBUTIONS ONLY option, you do not update index information. This option does not affect existing index information.

Use this option to avoid the examination of index information that can consume considerable processing time.

This option does not affect the recalculation of information on tables, such as the number of pages used, the number of rows, and fragment information. UPDATE STATISTICS needs this information to construct accurate column distributions and requires little time and system resources to collect it.

Do not confuse this DISTRIBUTIONS ONLY option with the DROP DISTRIBUTIONS ONLY option of LOW mode, whose syntax and semantics are not supported in MEDIUM or HIGH mode. For information on how to suppress the collection of column distributions, see Using the DROP DISTRIBUTIONS ONLY Option on page 893.

Using DBUPSPACE Settings to Suppress Index Information

You can also prevent indexes from being used by UPDATE STATISTICS operations in sorting rows by setting the third parameter of the **DBUPSPACE** environment variable to a value of 1. Refer to Chapter 3 of the *HCL® Informix® Guide to SQL: Reference* for information about the **DBSPACETEMP** and **DBUPSPACE** environment variables, which can restrict the system resources that are available for UPDATE STATISTICS operations. (The database server uses the storage locations that **DBSPACETEMP** specifies only when you use the HIGH option of UPDATE STATISTICS.)

Output for UPDATE STATISTICS from the SET EXPLAIN Statement

The SET EXPLAIN statement can display the plan that UPDATE STATISTICS uses to generate column distributions. The following output is based on the default **DBUPSPACE** value of 15 megabytes of sort memory, which in this example requires two passes to sort the 21.9 megabytes of data:

```
UPDATE STATISTICS:
==================

Table: zelaine.t1
Mode: HIGH
Number of Bins: 267 Bin size 2505
Sort data 21.9 MB Sort memory granted 15.0 MB
Estimated number of table scans 2
```

```
PASS #1 b
PASS #2 a
Scan 9 Sort 1 Build 2 Insert 0 Close 0 Total 12
Completed pass 1 in 0 minutes 12 seconds
Scan 5 Sort 2 Build 1 Insert 0 Close 0 Total 8
Completed pass 2 in 0 minutes 8 seconds
```

Routine Statistics

Before executing a new SPL routine for the first time, the database server optimizes the DML statements in the SPL routine. Immediately before you invoke that SPL routine, however, you can reduce the risk of error if you use the Routine Statistics syntax of UPDATE STATISTICS to update its query execution plans, some of which might reference tables whose schemas have been modified by DDL operations of concurrent sessions.

Optimization makes the code depend on the structure of tables referenced by the routine. If a DDL operation modifies the schema of a referenced table after the routine is optimized, but before it is executed, the routine can fail with an error.

This failure typically does not occur, however, if an index is added or dropped while automatic recompilation is enabled for routines referencing tables that ALTER TABLE, CREATE INDEX, or DROP INDEX operations have modified. This is the default behavior of Informix®. For more information about enabling or disabling automatic reoptimization after changes to the schema of a table, see the description of the IFX_AUTO_REPREPARE option to the SET ENVIRONMENT statement.

When the AUTO_REPREPARE configuration parameter and the IFX_AUTO_REPREPARE session environment variable are set to disable recompilation of SPL routines that reference tables whose schema has been modified, however, adding or dropping an index to a table that an SPL routine references indirectly can cause the routine to return error -710. To avoid this error after DDL operations, or to reoptimize SPL routines after table distributions might have been modified by DML operations, use the Routine Statistics segment of UPDATE STATISTICS to update the execution plans of any SPL routines that reference the table.

(explicit id unique_1272_Connect_42_rstat002) unique_1272_Connect_42_rstat002(explicit id unique_1272_Connect_42_rstat003)

unique_1272_Connect_42_rstat003

Routine Statistics

" **FOR** "

" { { **PROCEDURE** | { **FUNCTION** | **ROUTINE** } } [*routine* [([<Routine Parameter List>])]] | **SPECIFIC** { **PROCEDURE** | **FUNCTION** | **ROUTINE** } <Specific Name> }"

Element	Description	Restrictions	Syntax
owner	Owner of the UDR	No more than 32 bytes. Must be the authorization identifier of the owner of *routine*.	Owner name on page 1262

Element	Description	Restrictions	Syntax
routine	Name that a CREATE FUNCTION or CREATE PROCEDURE statement declared for an SPL routine	Must exist in the database. In ANSI-compliant databases, qualify *routine* with *owner* if you are not the *owner*.	Identifier on page 1231

Usage

If you change the structure of a table that an SPL routine references, you can run UPDATE STATISTICS FOR ROUTINE, FOR FUNCTION, or FOR PROCEDURE statements to reoptimize the routines that reference the table, rather than waiting until the next time an SPL routine that uses the table executes. (If a table that an SPL routine references is dropped, however, running UPDATE STATISTICS cannot prevent the SPL routine from failing with an error.)

The keywords of the Routine Statistics segment identify one or more SPL routine whose execution plan will be reoptimized.

Keyword

Which Execution Plans are Reoptimized

FUNCTION

The plan for any SPL function with the specified name (and with parameter types that match *routine parameter list,* if parameters are supplied). If you specify the FUNCTION keyword, the UPDATE STATISTICS statement fails with an error unless the specified routine returns a value or values, with or without the WITH RESUME option.

PROCEDURE

The plan for any SPL procedure with the specified name (and parameter types that match *routine parameter list,* if supplied)

ROUTINE

The plan for SPL functions and procedures with the specified name (and parameter types that match *routine parameter list,* if supplied)

SPECIFIC

The plan for the SPL routine called *specific name*. If you include the SPECIFIC keyword, the immediately following keyword must be either FUNCTION, PROCEDURE, or ROUTINE.

The parentheses symbols are optional if you omit the SPECIFIC keyword and include no argument list.

If you specify no routine name immediately after the FOR FUNCTION, FOR PROCEDURE, or FOR ROUTINE keywords, the execution plans are reoptimized for all SPL routines in the current database.

The database server keeps a list of tables that the SPL routine references explicitly. Whenever an explicitly referenced table is modified, the database server reoptimizes the procedure the next time the procedure is executed.

The **sysprocplan** system catalog table stores execution plans for SPL routines. Two actions can update the **sysprocplan** system catalog table:

- Execution of an SPL routine that uses a modified table
- The UPDATE STATISTICS FOR ROUTINE, FUNCTION, or PROCEDURE statement.

If you change a table that an SPL routine references, you can run UPDATE STATISTICS to reoptimize the procedures that reference the table, rather than waiting until the next time an SPL routine that uses the table executes. If a table that an SPL routine references is dropped, however, running UPDATE STATISTICS cannot prevent the SPL routine from failing with an error.

Examples of updating statistics for a specific routine

The following UPDATE STATISTICS FOR SPECIFIC statement instructs the database server to update statistics for an existing function named Perform_work that returns one or more values:

```
UPDATE STATISTICS FOR SPECIFIC FUNCTION Perform_work;
```

For the same Perform_work function, the effect of the following example is identical to that of the previous example:

```
UPDATE STATISTICS FOR SPECIFIC ROUTINE Perform_work;
```

Similarly, use the keywords SPECIFIC PROCEDURE or SPECIFIC ROUTINE to update statistics for SPECIFIC procedures that return no value.

Do not include parentheses or a parameter list after the name of the SPECIFIC routine. Because of the parentheses that follow the name of the Perform_work function, the following statement fails with an error:

```
UPDATE STATISTICS FOR SPECIFIC ROUTINE Perform_work();
```

The database server also issues an error if parentheses enclose arguments to the SPECIFIC routine, function, or procedure.

Altered tables that are referenced indirectly in SPL routines

After DDL operations that change the schema of a table that an SPL routine references indirectly, but not as the target of a DML operation, you might need to perform UPDATE STATISTICS operations on the modified table and on the SPL routine to avoid exceptions when the SPL routine is invoked.

If the SPL routine depends on a table that is referenced only indirectly, the database server cannot detect the need to reoptimize the procedure after that table is modified. For example, a table can be referenced indirectly if the SPL routine activates a trigger. After schema modifications of a table that is referenced by the trigger but not referenced directly by the SPL routine, the database server does not know that it should reoptimize execution plan of the SPL routine before running it. When the procedure is run after the table has been changed, error -710 can occur.

Each SPL routine is optimized the first time that it is run, not when it is created. This behavior means that an SPL routine might succeed the first time it is run, but might fail later under virtually identical circumstances, if the schema of an indirectly referenced table has been changed. The failure of an SPL routine can also be intermittent, because failure during one execution forces an internal warning to reoptimize the procedure before the next execution.

You can use either of two methods to recover from this error:

- Issue UPDATE STATISTICS FOR PROCEDURE to force reoptimization of the routine.
- Rerun the routine.

To prevent this error, you can force reoptimization of the SPL routine. To force reoptimization, execute the following statement:

```
UPDATE STATISTICS FOR PROCEDURE routine;
```

You can add this statement to your program in either of the following ways:

- Issue UPDATE STATISTICS after each statement that changes the mode of an object.
- Issue UPDATE STATISTICS FOR PROCEDURE before each invocation of the SPL routine.

For efficiency, you can put the UPDATE STATISTICS statement with the action that occurs less frequently in the program (change of object mode or execution of the procedure). In most cases, the action that occurs less frequently in the program is the change of object mode.

When you follow this method of recovering from this error, you must execute UPDATE STATISTICS FOR PROCEDURE for each SPL procedure that indirectly references the altered tables, unless the procedure also references the tables explicitly.

You can also recover from error -710 after an indirectly referenced table is altered simply by re-executing the SPL routine. The first time that the stored procedure fails, the database server marks the procedure as in need of reoptimization. The next time that you run the procedure, the database server reoptimizes the procedure before running it. Running the SPL routine twice, however, might be neither practical nor safe. A safer choice is to use the UPDATE STATISTICS FOR PROCEDURE statement to force reoptimization of the procedure.

When

Updating statistics when you upgrade the database server

When you upgrade a database to use with a newer database server, you can use the UPDATE STATISTICS statement to convert the indexes to the form that the newer database server uses.

You can choose to convert the indexes one table at a time or for the entire database at one time. Follow the conversion guidelines that are outlined in the *HCL Informix® Migration Guide*.

Migration to a newer database server is the only context in which the UPDATE STATISTICS statement causes table indexes to be implicitly dropped and re-created.

Performance considerations of UPDATE STATISTICS statements

The more specific you make the list of objects that the UPDATE STATISTICS statement examines, the faster it completes execution. Limiting the number of column distributions speeds the update. Similarly, precision affects the speed of the update. If all other keywords are the same, LOW works fastest, but HIGH examines the most data.

For version 11.70 and later of the Informix® database server, the AUTO_STAT_MODE setting can improve the efficiency of UPDATE STATISTICS operations that refresh data distribution statistics. This enables the database server to selectively recalculate only the table or fragment distributions that have become stale as a result of DML operations since the statistics

were last calculated, as determined by a change threshold that an explicit or default STATCHANGE table attribute defines. For information about how to set STATCHANGE and how to enable the automatic mode of UPDATE STATISTICS for refreshing only stale distribution statistics, see these topics:

- Using the FORCE and AUTO keywords on page 890
- AUTO_STAT_MODE session environment option
- #unique_1274
- Statistics options of the CREATE TABLE statement on page 368
- Statistics options of the ALTER TABLE statement on page 157

The USTLOW_SAMPLE environment option enables sampling during the gathering of index statistics for UPDATE STATISTICS operations in LOW mode. For an index with more than 100 K leaf pages, the gathering of statistics using sampling can increase the speed of the UPDATE STATISTICS operation.

Examples of UPDATE STATISTICS statements

```
UPDATE STATISTICS MEDIUM;
UPDATE STATISTICS MEDIUM RESOLUTION 10;
UPDATE STATISTICS MEDIUM RESOLUTION 10 .95;
{ RESOLUTION 10, CONFIDENCE .95}
UPDATE STATISTICS MEDIUM RESOLUTION 10 DISTRIBUTIONS ONLY;
UPDATE STATISTICS MEDIUM RESOLUTION 10 .95 DISTRIBUTIONS ONLY;

UPDATE STATISTICS HIGH;
UPDATE STATISTICS HIGH RESOLUTION 10;
UPDATE STATISTICS HIGH RESOLUTION 10 DISTRIBUTIONS ONLY;
```

Resolution must be greater than 0.005 and less than or equal to 10.0. Confidence must be in the range [0.80, 0.99] (inclusive).

Examples that follow are based on the **company_proc** procedure and **square_w_default** function, as defined below:

```
CREATE PROCEDURE company_proc ( no_of_items INT,
        itm_quantity SMALLINT, sale_amount MONEY,
        customer VARCHAR(50), sales_person VARCHAR(30) )
SPECIFIC spec_cmpy

DEFINE salesperson_proc VARCHAR(60);

-- Update the company table
INSERT INTO company_tbl VALUES (no_of_items, itm_quantity,
        sale_amount, customer, sales_person);

-- Generate the procedure name for the variable salesperson_proc
LET salesperson_proc = sales_person || "." || "tbl" ||
        month(current) || "_" || year(current) || "_proc" ;

-- Execute the SPL procedure that the salesperson_proc
-- variable specifies
EXECUTE PROCEDURE salesperson_proc (no_of_items,
        itm_quantity, sale_amount, customer);
END PROCEDURE;
```

```
CREATE FUNCTION square_w_default
        (i INT DEFAULT 0) {Specifies default value of i}
RETURNING INT {Specifies return of INT value}
SPECIFIC spec_square
        DEFINE j INT; {Defines routine variable j}
        LET j = i * i; {Finds square of i and assigns it to j}
        RETURN j; {Returns value of j to calling module}
END FUNCTION;
```

The UPDATE STATISTICS examples that follow reference the **company_proc** procedure and **square_w_default** function:

```
UPDATE STATISTICS  FOR PROCEDURE;
UPDATE STATISTICS  FOR PROCEDURE company_proc1;
UPDATE STATISTICS  FOR PROCEDURE
   company_proc1(INT,SMALLINT,MONEY,VARCHAR(50), VARCHAR(30));
UPDATE STATISTICS FOR SPECIFIC PROCEDURE spec_cmpy;

UPDATE STATISTICS FOR FUNCTION;
UPDATE STATISTICS FOR FUNCTION square_w_default;
UPDATE STATISTICS FOR FUNCTION square_w_default(INT);
UPDATE STATISTICS FOR SPECIFIC FUNCTION spec_square;
```

For a discussion of the performance implications of UPDATE STATISTICS, see your .

For a discussion of how to use the **dbschema** utility to view distributions created with UPDATE STATISTICS, see the *HCL Informix® Migration Guide*.

WHENEVER statement

Use the WHENEVER statement to trap exceptions that occur during the execution of SQL statements. The WHENEVER statement is equivalent to placing an exception-checking routine after every SQL statement.

Syntax

(explicit id unique_1293_Connect_42_wnvr001) unique_1293_Connect_42_wnvr001 **WHENEVER** { **SQLERROR** | **NOT FOUND** | [] **SQLWARNING** | [] **ERROR** } { **CONTINUE** | { **GOTO** | **GO TO** } { :*label* | [] *label* } | **CALL** *routine* | **STOP** }

Element	Description	Restrictions	Syntax
label	Statement label to which program control transfers when an exception occurs	Must exist in the same source-code module.	Language-specific
routine	Name of a user-defined routine (UDR) to be invoked when an exception occurs	No arguments; UDR must exist at compile time.	Identifier on page 1231

Usage

Important: Use this statement only with .

The following table summarizes the types of exceptions that you can check with the WHENEVER statement.

Type of exception	WHENEVER Keyword	For More Information
Errors	SQLERROR or ERROR	SQLERROR Keyword on page 906
Warnings	SQLWARNING Keyword on page 906	
Not Found or End of Data	NOT FOUND Keywords on page 907	

Programs that do not use the WHENEVER statement do not automatically abort when an exception occurs. Such programs must explicitly check for exceptions and take whatever corrective action their logic specifies. If you do not check for exceptions, the program simply continues running. If errors occur, however, the program might not perform its intended purpose.

The first keyword that follows WHENEVER specifies some type of exceptional condition; the last part of the statement specifies some action to take when the exception is encountered (or no action, if CONTINUE is specified). The following table summarizes possible actions that WHENEVER can specify.

Type of action	WHENEVER keyword	For more information
Continue program execution	CONTINUE Keyword on page 907	
Stop program execution	STOP Keyword on page 907	
Transfer control to a specified label	GOTO GO TO	GOTO Keyword on page 907
Transfer control to a UDR	CALL Clause on page 908	

The Scope of WHENEVER

WHENEVER is a preprocessor directive, rather than an executable statement. The preprocessor, not the database server, handles the interpretation of the WHENEVER statement. When the preprocessor encounters a WHENEVER statement in the source file, it inserts appropriate code into the preprocessed code after each SQL statement, based on the exception and the action that WHENEVER specifies. The scope of the WHENEVER statement begins where the statement appears in the source module and remains in effect until the preprocessor encounters one or the other of the following things while sequentially processing the source module:

- The next WHENEVER statement with the same condition (SQLERROR, SQLWARNING, or NOT FOUND) in the same source module
- The end of the source module

The following example program has three WHENEVER statements, two of which are WHENEVER SQLERROR statements. Line 4 uses STOP with SQLERROR to override the default CONTINUE action for errors.

Line 8 specifies the CONTINUE keyword to return the handling of errors to the default behavior. For all SQL statements between lines 4 and 8, the preprocessor inserts code that checks for errors and halts program execution if an error occurs. Therefore, any errors that the INSERT statement on line 6 generates cause the program to stop.

After line 8, the preprocessor does not insert code to check for errors after SQL statements. Therefore, any errors that the INSERT statement (line 10), the SELECT statement (line 11), and DISCONNECT statement (line 12) generate are ignored. The SELECT statement, however, does not stop program execution if it does not locate any rows; the WHENEVER statement on line 7 tells the program to continue if such an exception occurs:

```
1   main()
2   {
3   EXEC SQL connect to 'test';
4   EXEC SQL WHENEVER SQLERROR STOP;
5   printf("\n\nGoing to try first insert\n\n");
6   EXEC SQL insert into test_color values ('green');
7   EXEC SQL WHENEVER NOT FOUND CONTINUE;
8   EXEC SQL WHENEVER SQLERROR CONTINUE;
9   printf("\n\nGoing to try second insert\n\n");
10   EXEC SQL insert into test_color values ('blue');
11   EXEC SQL select paint_type from paint where color='red';
12   EXEC SQL disconnect all;
13   printf("\n\nProgram over\n\n");
14   }
```

SQLERROR Keyword

If you use the SQLERROR keyword, any SQL statement that encounters an error is handled as the WHENEVER SQLERROR statement directs. If an error occurs, the **sqlcode** variable (**sqlca.sqlcode**, **SQLCODE**) is set to a value less than zero (`0`) and the SQLSTATE variable is set to a class code with a value greater than `02`.

The next example terminates program execution if an SQL error is detected:

```
WHENEVER SQLERROR STOP
```

If you do not include any WHENEVER SQLERROR statements in a program, the default action for WHENEVER SQLERROR is CONTINUE.

ERROR Keyword

Within the WHENEVER statement (and only in this context), the keyword ERROR is a synonym for the SQLERROR keyword.

SQLWARNING Keyword

If you use the SQLWARNING keyword, any SQL statement that generates a warning is handled as the WHENEVER SQLWARNING statement directs. If a warning occurs, the first field (**sqlca.sqlwarn.sqlwarn0**) of the warning structure in SQLCA is set to `W`, and the SQLSTATE variable is set to a class code of `01`.

Besides the first field of the warning structure, a warning also sets an additional field to `W`. The field that is set indicates what type of warning occurred.

The next statement causes execution to stop if a warning condition exists:

```
WHENEVER SQLWARNING STOP
```

If you do not use any WHENEVER SQLWARNING statements in a program, the default action for WHENEVER SQLWARNING is CONTINUE.

NOT FOUND Keywords

If you use the NOT FOUND keywords, exception handling for SELECT and FETCH statements (including implicit SELECT and FETCH statements in FOREACH and UNLOAD statements) is treated differently from other SQL statements. The NOT FOUND keyword checks for the following cases:

- The **End of Data** condition: a FETCH statement that attempts to get a row beyond the first or last row in the active set
- The **Not Found** condition: a SELECT statement that returns no rows

In each case, the **sqlcode** variable is set to `100`, and the SQLSTATE variable has a class code of `02`. For the name of the **sqlcode** variable in each HCL® Informix® product, see the table in SQLERROR Keyword on page 906.

The following statement calls the **no_rows( )** function each time the NOT FOUND condition exists:

```
WHENEVER NOT FOUND CALL no_rows
```

If you do not use any WHENEVER NOT FOUND statements in a program, the default action for WHENEVER NOT FOUND is CONTINUE.

CONTINUE Keyword

Use the CONTINUE keyword to instruct the program to ignore the exception and to continue execution at the next statement after the SQL statement. The default action for all exceptions is CONTINUE. You can use this keyword to turn off a previously specified action for an exceptional condition.

STOP Keyword

Use the STOP keyword to instruct the program to stop execution when the specified exception occurs. The following statement halts execution of an ESQL/C program each time that an SQL statement generates a warning:

```
EXEC SQL WHENEVER SQLWARNING STOP;
```

GOTO Keyword

Use the GOTO clause to transfer control to the statement that the label identifies when a specified exception occurs. The GOTO and GO TO keywords are ANSI-compliant syntax for this feature of embedded SQL languages like ESQL/C. The following code fragment shows a WHENEVER statement that transfers control to the label **missing** each time that the NOT FOUND condition occurs:

```
query_data()
   ...
   EXEC SQL WHENEVER NOT FOUND GO TO missing;
```

```
   ...
   EXEC SQL fetch lname into :lname;
   ...
   missing:
      printf("No Customers Found\n");
```

Within the scope of the WHENEVER GOTO statement, you must define the labeled statement in *each* routine that contains SQL statements. If your program contains more than one user-defined function, you might need to include the labeled statement and its code in *each* function.

If the preprocessor encounters an SQL statement within the scope of a WHENEVER ... GOTO statement, but within a routine that does not have the specified label, the preprocessor tries to insert the code associated with the labeled statement, but generates an error when it cannot find the label.

To correct this error, either put a labeled statement with the same label name in each UDR, or issue another WHENEVER statement to reset the error condition, or use the CALL clause to call a separate function.

CALL Clause

Use the CALL clause to transfer program control to the specified UDR when the specified type of exception occurs. Do not include parentheses after the UDR name. The following WHENEVER statement causes the program to call the **error_recovery()** function if the program detects an error:

```
EXEC SQL WHENEVER SQLERROR CALL error_recovery;
```

When the UDR returns, execution resumes at the next statement after the line that is causing the error. If you want to halt execution when an error occurs, include statements that terminate the program as part of the specified UDR.

Observe the following restrictions on the specified routine:

- The UDR cannot accept arguments nor can it return values. If it needs external information, use global variables or the WHENEVER ... GOTO option to transfer program control to a label that calls the UDR.
- You cannot specify the name of an SPL routine in the CALL clause. To call an SPL routine, use the CALL clause to invoke a UDR that contains the EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement.
- Make sure that all functions within the scope of WHENEVER ... CALL statements can find a declaration of the specified function.

Related Statements

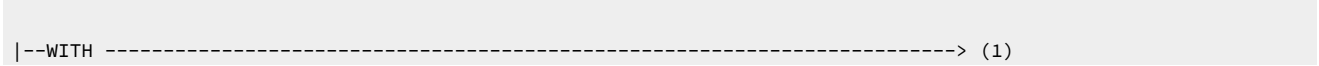
Related statements: EXECUTE FUNCTION statement on page 502, EXECUTE PROCEDURE statement on page 510, and FETCH statement on page 513

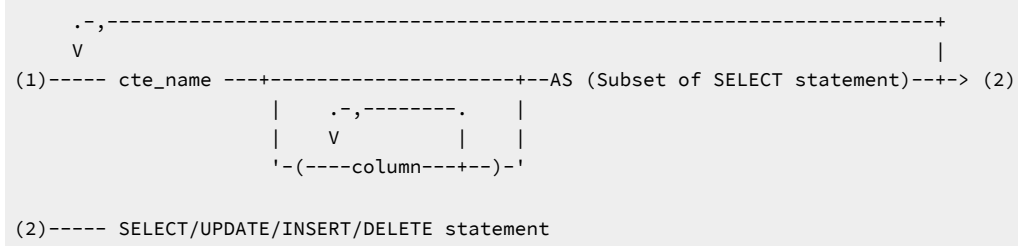
For discussions on exception handling and error checking, see the *HCL® Informix® Enterprise Replication Guide*.

WITH statement (Common Table Expressions)

A common table expression (CTE) is a named temporary result set that exists within the scope of a single statement and that can be referred to later within that statement, possibly multiple times. The following discussion describes how to write statements that use CTEs.

Syntax





Element	Description	Restrictions	Syntax
column	Name that you declare here for a column in CTE. Default is a column name from Projection list of SELECT		Identifier
cte_name	Name that you declare here for the table expression, and it can be used as a virtual table in the following SELECT/UPDATE/INSERT/DELETE statement	Must be unique among view, table, sequence, and synonym names in the database.	Identifier

Usage

WITH statement can be run directly like SELECT/UPDATE/INSERT/DELETE statements, and it can be used with

- CREATE VIEW
- CREATE TRIGGER
- CREATE PROCEDURE/FUNCTION

Recursive CTE

A recursive CTE starts with either one non-recursive sub-query or several non-recursive sub-queries joined by UNION or UNION ALL and ends with exactly one recursive sub-query joined by UNION ALL. A recursive sub-query references the CTE being defined.

```
WITH recursive_cte AS (
            <initial subquery>
            UNION ALL
            <recursive subquery>
) SELECT …
```

Example

Example 1: Recursive query computing the factorial of numbers from 0 to 9

```
WITH temp (n, fact) AS (
 SELECT 0, 1                                          -- Initial Subquery
  UNION ALL
 SELECT n+1, (n+1)*fact FROM temp       -- Recursive Subquery
        WHERE n < 9)
SELECT * FROM temp;
```

```
Other example of computing Fibonacci Numbers
WITH fib(p, n) as (
    select 0, 1                              -- initial subquery
    UNION ALL                               -- 'UNION ALL'
    select n, (p+n) from fib -- recursive subquery
             where n < 100                -- terminate condition
)
select p as fn from fib;
        fn
          0
          1
          1
          2
          …
         34
         55
         89
```

Example

Example 2: Recursive CTE optional Cycle clause

```
CYCLE <column list> SET <cycle pseudo column>
         TO <value1> DEFAULT <value2>

create table cycle (id int, pid int);
insert into cycle values (1,2);
insert into cycle values (2,1);

WITH cte AS (
    select id, pid from cycle where id = 1 UNION ALL
    select t.id, t.pid
               from cycle t, cte where t.pid = cte.id)
cycle id set iscycle to "yes" default "no"
SELECT id, pid, iscycle from cte ;
             id         pid iscycle
          1           2 no
          2           1 yes
```

Example

Example 3: Recursive CTE Loops

```
with cte(n) as
(
    select 1
    UNION ALL
    select n+1 from cte
)
select first 2 n from cte;            -- 2 rows return

without the 'FIRST 2' the query will loop till integer limit error.
```

SPL statements

These topics describe Stored Procedure Language (SPL) statements, which you use to write SPL routines. You can store these routines in the database as user-defined routines (UDRs).

SPL routines (formerly referred to as *stored procedures*) are effective tools for controlling SQL activity. This chapter contains descriptions of the SPL statements. The description of each statement includes the following information:

- A brief introduction that explains the effect of the statement
- A syntax diagram that shows how to enter the statement correctly
- A syntax table that explains each input parameter in the syntax diagram
- Rules of usage, including examples that illustrate these rules

If a statement is composed of multiple clauses, the statement description provides information about each clause.

For an overview of the SPL language and task-oriented information about creating and using SPL routines, see the *HCL® Informix® Guide to SQL: Tutorial*.

For an overview with detailed examples of how to create and use prepared objects and Dynamic SQL in SPL routines, see this IBM® developerWorks® article: Dynamic SQL support in Informix® Dynamic Server Stored Procedure Language.

Informix® can create an SPL function with the CREATE PROCEDURE or CREATE PROCEDURE FROM statement, but requires the CREATE FUNCTION or CREATE FUNCTION FROM statement for external functions. It is recommended, however, that you use the CREATE FUNCTION or CREATE FUNCTION FROM statement to create new user-defined functions.

Debugging SPL routines

You can use a Routine Debugger client application to identify and analyze logical errors in an SPL routine.

You can include the TRACE statement in your SPL routine to generate tracing output while the SPL routine runs. For information about how to produce and examine TRACE statement output, see the description of TRACE on page 959.

Current® restrictions for debugging SPL routines

The following software products currently support client environments that can be used for debugging Informix® SPL routines:

- Optim™ Development Studio (ODS)
- IBM® Database Add-Ins for Visual Studio (IDAIVS)

Refer to the data type support document for information on which Informix® data types are read-only and which are updateable.

The following restrictions apply to both the Optim™ Development Studio (ODS) and IBM® Database Add-Ins for Visual Studio (IDAIVS) debugging environments:

Unlogged databases

Informix® nontransactional databases do not support SPL debugging. You cannot use either the ODS or the IDAIVS debugging environments with the Informix® database whose CREATE DATABASE statement omits the WITH LOG keywords.

Secondary server instances

Secondary servers within a cluster environment do not support 'Step into' trigger procedure operation for Insert, Delete, or Update triggers.

Data strings delimited by double (") quotation marks

The current interpretation by the ODS or IDAIVS debugging client of quotation-mark delimiters might conflict with the expected behavior of some SPL routines.

- Currently, in ODS debugging sessions the **DELIMIDENT** environment variable is set to `'y'` by default in the HCL Data Server Driver for JDBC and SQLJ connection string. The default value of **DELIMIDENT** is `'n'` for JDBC connection.
- Currently in IDAIVS debugging sessions, the **DELIMIDENT** environment variable is set to `'y'` by default in the connection string.

The setting of the **DELIMIDENT** environment variable affects how the database server interprets quoted strings:

- `'y'` specifies that strings enclosed by double quotation (`"`) marks are delimited SQL identifiers. Client applications must use single quotation (`'`) marks to delimit character strings, and must use double quote (`"`) symbols only around delimited SQL identifiers, which can support a larger character set than is valid in undelimited identifiers. In locales that support lettercase sensitivity, letters within delimited strings or within delimited identifiers are case-sensitive. This is the default value for .NET.
- `'n'` specifies that client applications can use double (`"`) or single (`'`) quotation marks to delimit character strings, but not to delimit SQL identifiers. If the database server encounters a string that is delimited by double or single quotation marks in a context where an SQL identifier is required, it issues an error. An owner name that qualifies an SQL identifier can be delimited by single quotation (`'`) marks. You must use a pair of the same quotation mark symbols to delimit a character string.
- If no value for **DELIMIDENT** is specified on the client system, the default setting is used. As indicated above, for OCD this default is `'n'` (from the HCL Data Server Driver for JDBC and SQLJ connection string. For IDAIVS. this default is `'y'` (from the connection string).

Restrictions that only affect ODS debugging sessions

By default, 'AUTOCOMMIT' transaction mode is set to 'TRUE' by the HCL Data Server Driver for JDBC and SQLJ. This is enabled in ODS debugging session for all logging databases, including databases that were created WITH LOG MODE ANSI, and also databases that are not ANSI-compliant.

Currently, the Informix® server performs an implicit commit operation after every SQL statement completes in an Optim™ Development Studio debugging session. If there is an explicit transaction started inside the Informix® SPL procedure, the database server ignores SQL ERROR -535 and continues the debugging session.

Restrictions that only affect IDAIVS debugging sessions

Currently, SPL function debugging is not supported by IBM® Database Add-Ins for Visual Studio. You can only use the IDAIVS debugging environment with Informix® SPL procedures that do not return any value to the calling context.

<< *Label* >> statement

Use the <<*label*>> statement of SPL to declare a statement label or a loop label.

- A *statement label* is an SQL identifier, delimited by double angle-brackets, immediately preceding a statement within a statement block to which the GOTO statement of SPL can transfer control of program execution.
- A *loop label* is an SQL identifier, delimited by double angle-brackets, immediately preceding a loop statement of SPL. The same label, without double angle-bracket delimiters, can follow the END LOOP keywords, or END FOR keywords, or END WHILE keywords that terminate the labeled loop. The EXIT *label* statement can pass control of program execution to whatever statement immediately follows the undelimited loop label.

Note that *label* is not a keyword of the <<*label*>> statement, but is a placeholder for some specific user-defined identifier of the statement label or loop label that the <<*label*>> statement declares.

Syntax

```
<<label>>
```

Element	Description	Restrictions	Syntax
label	Name that you declare here for a statement label or for a loop label	Must be unique among the identifiers of statement labels and of loop labels within the SPL routine	Identifier on page 1231

Usage

You can use the <<*Label*>> statement in two ways:

- To declare a *statement label* before an executable statement to which the GOTO statement of SPL can transfer control of execution. The SPL statement that immediately follows the statement label declaration is called a *labeled statement*.
- To declare a *loop label* immediately before a LOOP, FOR, or WHILE statement of SPL. The LOOP, FOR, or WHILE statement that immediately follows the loop label declaration is called a *labeled loop*.

The EXIT *label* or EXIT *label* WHEN (*condition*) statement can exit from the labeled loop, passing control of execution to the statement immediately following an END LOOP *label* statement. The *label* specified in the EXIT statement can match

the label identifier of the labeled loop of the EXIT statement, or if loops are nested, this *label* can match the label of an outer labeled loop. In either case, the EXIT *label* statement passes control to a statement that follows an END LOOP *label* statement that specifies the same loop label. This EXIT *label* behavior differs from that of the GOTO *label* statement, which passes control to the statement that follows the declaration of the specified statement label.

The following restrictions apply to labels in SPL routines:

- The name of a statement label must be within the scope of reference of the GOTO statement.
- The GOTO option of the WHENEVER statement of SQL cannot reference an SPL statement label, because the WHENEVER statement is valid only in ESQL/C applications.
- The GOTO statement of SPL cannot reference a loop label.
- The GOTO statement cannot reference a statement label within an ON EXCEPTION statement block.
- A statement label cannot be declared within an ON EXCEPTION statement block.
- The label name must be unique among statement labels and loop labels within the SPL routine.

Examples of Labels

The following example illustrates a statement label called **increment_x** within an SPL routine:

```
DEFINE x INT;
LET x = 0;
BEGIN
        <<increment_x>>
        BEGIN
                LET x = x + 1;
        END;
        IF x < 10 THEN
                GOTO increment_x;
        END IF;
END;
END  PROCEDURE;
```

The following program fragment shows an example of a labeled FOR loop:

```
<<lb_for>>
FOR i IN 1..5
   i := i +1 ;
END FOR lb_for;
```

The following program fragment illustrates a labeled loop from which an EXIT *label* statement can exit:

```
<<outer>>
LOOP
...
LOOP
...
EXIT outer WHEN ... -- exit from both loops
END LOOP;
...
END LOOP outer;
```

Related Statements

EXIT on page 929, FOR on page 931, GOTO on page 940, LOOP on page 945, WHILE on page 961

CALL

Use the CALL statement to execute a user-defined routine (UDR) from within an SPL routine.

Syntax

(explicit id unique_1313_Connect_42_call002) unique_1313_Connect_42_call002 **CALL** { *procedure* ([<Argument>[]]) | *function* ([<Argument>[]]) **RETURNING** *data_var* | *routine_var* [**RETURNING** *data_var*] }

Element	Description	Restrictions	Syntax
data_var	Variable to receive the values *function* returns	The data type of *data_var* must be appropriate for the returned value	Identifier on page 1231
function, *procedure*	User-defined function or procedure	The function or procedure must exist	Identifier on page 1231
routine_var	Variable that contains the name of a UDR	Must be a character data type that contains the non-NULL name of an existing UDR	Identifier on page 1231

Usage

The CALL statement invokes a UDR. The CALL statement is identical in behavior to the EXECUTE PROCEDURE and EXECUTE FUNCTION statements, but you can only use CALL from within an SPL routine.

You can use CALL in programs or with , but only if the statement is in an SPL routine that the program or executes.

When you use CALL to invoke a user-defined function that you specify by its *function* identifier or as a *routine_var* that stores the identifier of the function, the CALL statement must also include the RETURNING clause.

The CALL statement cannot invoke an iterator TABLE function from a subquery in the FROM clause of a SELECT statement. For the syntax of iterator TABLE functions, see Iterator Functions on page 720.

Specifying Arguments

The argument list of the CALL statement, delimited by parentheses, immediately follows the name of the UDR. If you include no arguments, empty parentheses must follow the name of the UDR. If the list includes more arguments than the UDR expects, you receive an error.

If CALL specifies fewer arguments than the UDR expects, the arguments are said to be missing. The database server initializes missing arguments to their corresponding default values. (See CREATE PROCEDURE and CREATE FUNCTION.) This initialization occurs before the first executable statement in the body of the UDR. If missing arguments do not have default values, they are initialized to the value of UNDEFINED. An attempt to use any variable of UNDEFINED value results in an error.

In each UDR call, you have the option of specifying parameter names for the arguments that you pass to the UDR. Each of the following examples are valid for a UDR that expects character arguments named `t`, `n`, and `d`, in that order:

```
CALL add_col (t='customer', n = 'newint', d ='integer');
CALL add_col('customer','newint','integer');
```

Both of the CALL statements above have the same effect.

The syntax of the argument list is described in more detail in the topic Arguments on page 1209.

Receiving input from the called UDR

The RETURNING clause specifies the variable that receives values that the function returns to its calling context.

The following example shows two UDR calls:

```
CREATE PROCEDURE not_much()
   DEFINE i, j, k INT;
   CALL no_args (10,20);
   CALL yes_args (5) RETURNING i, j, k;
END PROCEDURE;
```

The first routine call (**no_args**) expects no returned values. The second routine call is to a function (**yes_args**), which expects three returned values. The **not_much()** procedure declares three integer variables (**i**, **j**, and **k**) to receive the returned values from **yes_args**.

CASE

Use the CASE statement when program execution needs to take exactly one among multiple branches, depending on the value of an SPL variable or of a simple expression. The CASE statement is a logical alternative to the IF statement.

Syntax

(explicit id unique_1316_Connect_42_case002) unique_1316_Connect_42_case002 **CASE** *value_expr* { <ELSE Clause> | **WHEN** { *constant_expr* | **NULL** } **THEN** <Statement Block>[] [<ELSE Clause>] } **END CASE**

ELSE Clause

" **ELSE** <Statement Block>[] "

Element	Description	Restrictions	Syntax
constant_expr	Expression that specifies a literal value	Can be a literal number, quoted string, literal datetime, or literal interval. The data type must be compatible with the data type of *value_expr*.	Constant Expressions on page 1040
value_expr	Expression that returns a value	An SPL variable or any other type of expression that returns a value or NULL. The data type cannot be a large object (BLOB, BYTE, CLOB, TEXT), a collection, nor a user-defined OPAQUE or	Expression on page 1003

Element	Description	Restrictions	Syntax
		DISTINCT type. Among built-in OPAQUE types, only BOOLEAN and LVARCHAR are valid.	

Usage

You can use the CASE statement to create a set of conditional branches within an SPL routine. The WHEN and the ELSE clauses are optional, but you must include at least one of them.

- If you include no WHEN clause and no ELSE clause, the CASE statement fails with a syntax error.
- If you include no ELSE clause, but no WHEN clause specifies a *constant expr* that matches the *value_expr*, the CASE statement fails with runtime error -26062 when the routine is executed.

Do not confuse the CASE statement with CASE expressions of SQL. (CASE expressions support the same keywords as the CASE statement, but use different syntax and semantics to evaluate *conditions* that you specify. CASE expressions return a single value or NULL, as described in CASE Expressions on page 1032.)

How the database server executes a CASE statement

The database server executes the CASE statement by taking the following sequence of actions:

- The database server evaluates the *value_expr* expression.
- If the resulting value matches a literal value specified as the *constant_expr* specification of a WHEN clause, the database server executes the statement block that immediately follows the THEN keyword in that WHEN clause.
- If the value to which the *value_expr* parameter evaluates matches the *constant_expr* specification in more than one WHEN clause, the database server executes the statement block that immediately follows the THEN keyword in the first matching WHEN clause of the CASE statement. (In this case, the lexical order of the WHEN clauses can determine the result of the CASE statement.) If the database server executes a GOTO statement in the statement block that follows the THEN keyword, the database server transfers program control to the specified statement label. Otherwise, the database server executes the next SPL or SQL statement that follows the END CASE keywords that mark the end of the current CASE statement in the SPL routine.
- If the value to which the *value_expr* parameter evaluates does not match the literal value specified in the *constant_expr* specification of any WHEN clause, and if the CASE statement includes an ELSE clause, the database server executes the statement block that immediately follows the ELSE keyword. If the database server executes a GOTO statement in the statement block that follows the ELSE keyword, the database server transfers program control to the specified statement label. Otherwise, the database server executes the next SPL or SQL statement that follows the END CASE keywords that mark the end of the current CASE statement in the SPL routine.
- If the value to which the *value_expr* parameter evaluates does not match the literal value specified in the *constant_expr* specification of any WHEN clause, and if the CASE statement does not includes an ELSE clause, the database server issues an exception, and the CASE statement fails with error -26062. Whether the SPL routine terminates or continues to execute depends on its exception-handling logic.

This implementation of the CASE statement closely resembles that of HCL Informix® Parallel Server, except that HCL Informix® Parallel Server issues no error when no ELSE clause is specified and no WHEN clause matches the *value_expr* parameter. In this case, program execution continues at the SPL or SQL statement that immediately follows the CASE statement.

The statement block that follows the THEN or ELSE keywords can include any SQL statement or SPL statement that is valid in a statement block of an SPL routine. For more information, see Statement Block on page 1289.

Evaluating the value expression in a CASE statement

The database server calculates the value of the *value_expr* parameter only once, at the start of execution of the CASE statement. If the expression specified in that parameter contains one or more SPL variables, and the value of any of these variables subsequently changes in one of the statement blocks within the CASE statement, the database server does not recalculate the value of the *value_expr* parameter. For this reason, any change in the value of variables specified in the *value_expr* parameter has no effect on the branch taken by the CASE statement.

Examples of CASE statements

In the following example, the CASE statement initializes one of a set of SPL variables (named **j**, **k**, **l**, and **m**) to the value of an SPL variable named **x**, depending on the value of another SPL variable named **i**:

```
CASE i
   WHEN 1 THEN LET j = x;
   WHEN 2 THEN LET k = x;
   WHEN 3 THEN LET l = x;
   WHEN 4 THEN LET m = x;
   ELSE
      RAISE EXCEPTION 100; --invalid value
END CASE;
```

Here each WHEN clause specifies a literal integer as its constant expression, under the assumption that the value expression has a numeric data type. (If these literal values had been delimited by quotation marks, the database server would treat them as character values.)

The following example includes a test for NULL in a WHEN clause, where the value expression and constant expression data types are CHAR(1):

```
CREATE PROCEDURE case_proc( )
RETURNING CHAR(1);
DEFINE grade CHAR(1);
LET grade = 'D';
CASE grade
   WHEN 'A' THEN LET grade = 'a';
   WHEN 'B' THEN LET grade = 'b';
   WHEN 'C' THEN LET grade = 'c';
   WHEN NULL THEN LET grade = 'z';
   ELSE LET grade = 'd';
END CASE;
RETURN grade;
END PROCEDURE;
```

Related statement

IF on page 941

CONTINUE

Use the CONTINUE statement to start the next iteration of the innermost FOR, LOOP, WHILE, or FOREACH loop.

Syntax

```
CONTINUE [ { FOR | FOREACH | LOOP | WHILE } ] ;
```

Usage

When control of execution passes to a CONTINUE statement, the SPL routine skips the rest of the statements in the innermost loop of the specified type. Execution continues at the top of the loop with the next iteration.

In the following example, the **loop_skip** function inserts values `3` through `15` into the table **testtable**. The function also returns values `3` through `9` and `13` through `15` in the process. The function does not return the value `11` because it encounters the CONTINUE FOR statement. The CONTINUE FOR statement causes the function to skip the RETURN WITH RESUME statement:

```
CREATE FUNCTION loop_skip()
   RETURNING INT;
   DEFINE i INT;
   ...
   FOR i IN (3 TO 15 STEP 2)
      INSERT INTO testtable values(i, null, null);
      IF i = 11
         CONTINUE FOR;
      END IF;
      RETURN i WITH RESUME;
   END FOR;

END FUNCTION;
```

Just as with the EXIT statement, (EXIT on page 929), in FOREACH statements and in FOR or WHILE statements that do not include the LOOP keyword, the FOR, WHILE, or FOREACH keyword must immediately follow the CONTINUE keyword to specify the type of loop. Errors are generated if the specified type of loop does not match the context in which the CONTINUE statement is issued.

In the LOOP, FOR LOOP, and WHILE LOOP statements, whether labeled or unlabeled, a keyword indicating the type of loop is optional after the CONTINUE keyword, but Informix® issues an error if you specify a keyword that does not correspond to the type of loop.

Example

Related Statements

FOR on page 931, FOREACH on page 935, LOOP on page 945, WHILE on page 961

DEFINE

Use the DEFINE statement to declare local variables that an SPL routine uses, or to declare global variables that can be shared by several SPL routines.

Syntax

(explicit id unique_1320_Connect_42_def003) unique_1320_Connect_42_def003(explicit id unique_1320_Connect_42_def004) unique_1320_Connect_42_def004

DEFINE { **GLOBAL** *SPL_var* { *data_type***DEFAULT** <Default Value>[] | **REFERENCES** { **BYTE** | **TEXT** } **DEFAULT NULL** } | *SPL_var* { *data_type* | **REFERENCES** { **BYTE** | **TEXT** } | **LIKE** { *view* | *synonym* | *table* } .*column* | { **PROCEDURE** } | { { **BLOB** | **CLOB** } | { <Subset of Complex Data Types>[] | *distinct_type* | *opaque_type* } } } } ;

Element	Description	Restrictions	Syntax
column	Column name	Must already exist in the *table* or *view*	Identifier on page 1231
data_type	Type of *SPL_var*	See Declaring Global Variables on page 921	Data Type on page 985
distinct_type	A distinct type	Must already be defined in the database	Data Type on page 985
opaque_type	An opaque type	Must already be defined in the database	Data Type on page 985
SPL_var	New SPL variable	Must be unique within statement block	Identifier on page 1231
synonym, *table*, *view*	Name of a table, view, or synonym	Synonym and the table or view to which it points must exist when DEFINE is issued	Identifier on page 1231

Usage

The DEFINE statement is not an executable statement. The DEFINE statement must appear after the routine header and before any other statements. If you declare a local variable (by using DEFINE without the GLOBAL keyword), its scope of reference is the statement block in which it is defined. You can use the variable within the statement block. Another variable outside the statement block with a different definition can have the same name.

A variable with the GLOBAL keyword is global in scope and is available outside the statement block and to other SPL routines. Global variables can be any built-in data type except BIGSERIAL, BLOB, BYTE, CLOB, SERIAL, SERIAL8, or TEXT. Local variables can be any built-in data type except BIGSERIAL, BYTE, SERIAL, SERIAL8, or TEXT. If *column* is of the BIGSERIAL, SERIAL, or SERIAL8 data type, declare a BIGINT, INT, or INT8 variable to store its value.

Declaring the names of SQL keywords or the identifiers of other database objects as SPL variables can produce errors or unexpected results in some contexts. For discussions of some potential problems of name conflicts that involve SPL variables, see the related concepts below.

Referencing TEXT and BYTE Variables

The REFERENCES keyword lets you use BYTE and TEXT variables. These do not contain the actual data but are pointers to the data. The REFERENCES keyword indicates that the SPL variable is just a pointer. You can use BYTE and TEXT variables exactly as you would use any other variable in SPL.

Redeclaration or Redefinition

If you define the same variable twice in the same statement block, you receive an error. You can redefine a variable within a nested block, in which case it temporarily hides the outer declaration. This example produces an error:

```
CREATE PROCEDURE example1()
   DEFINE n INT; DEFINE j INT;
   DEFINE n CHAR (1);    -- redefinition produces an error
```

Redeclaration is valid in the following example. Within the nested statement block, **n** is a character variable. Outside the block, **n** is an integer variable.

```
CREATE PROCEDURE example2()
   DEFINE n INT; DEFINE j INT;
   ...
   BEGIN
   DEFINE n CHAR (1);    -- character n masks global integer variable
   ...
END;
```

Declaring Global Variables

Use the following syntax for declaring global variables:

(explicit id unique_1323_Connect_42_dgv002) unique_1323_Connect_42_dgv002 **DEFINEGLOBAL** *SPL_var* { *data_type***DEFAULT** <Default Value>[] | **REFERENCES** { **BYTE** | **TEXT** } **DEFAULT NULL** } ;

Element	Description	Restrictions	Syntax
data_type	Type of *SPL_var*	See Declaring Global Variables on page 921.	Data Type on page 985
SPL_var	New SPL variable	Must be unique within statement block	Identifier on page 1231

The GLOBAL keyword indicates that the variables that follow have a scope of reference that includes all SPL routines that run in a given DB-Access or SQL administration API session. The data types of these variables must match the data types of variables in the *global environment*. The global environment is the memory that is used by all the SPL routines that run in a given DB-Access or SQL administration API session. The values of global variables are stored in memory.

SPL routines that are running in the current session share global variables. Because the database server does not save global variables in the database, the global variables do not remain when the current session closes.

The first declaration of a global variable establishes the variable in the global environment; subsequent global declarations simply bind the variable to the global environment and establish the value of the variable at that point.

The following example shows two SPL procedures, **proc1** and **proc2**; each has defined the global variable **gl_out**:

- SPL procedure **proc1**

```
CREATE PROCEDURE proc1()
   ...
   DEFINE GLOBAL gl_out INT DEFAULT 13;
   ...
   LET gl_out = gl_out + 1;
END PROCEDURE;
```

- SPL procedure **proc2**

```
CREATE PROCEDURE proc2()
   ...
   DEFINE GLOBAL gl_out INT DEFAULT 23;
   DEFINE tmp INT;
   ...
   LET tmp = gl_out
END PROCEDURE;
```

If **proc1** is called first, **gl_out** is set to `13` and then incremented to `14`. If **proc2** is then called, it sees that **gl_out** is already defined, so the default value of `23` is not applied. Then, **proc2** assigns the existing value of `14` to **tmp**. If **proc2** had been called first, **gl_out** would have been set to `23`, and `23` would have been assigned to **tmp**. Later calls to **proc1** would not apply the default of `13`.

Databases of different database server instances do not share global variables, but all the databases of the same database server instance can share global SPL variables in a single session. The database server and any application development tools, however, do not share global variables.

Default Value

Global variables can have literal, NULL, or system constant default values.

(explicit id unique_1321_Connect_42_defv001) unique_1321_Connect_42_defv001(explicit id unique_1321_Connect_42_defv002)

unique_1321_Connect_42_defv002(explicit id unique_1321_Connect_42_defv003) unique_1321_Connect_42_defv003(explicit id

unique_1321_Connect_42_defv004) unique_1321_Connect_42_defv004(explicit id unique_1321_Connect_42_defv005) unique_1321_Connect_42_defv005

Default Value

" { <Literal Number> [] | <Quoted String> [] | <Literal Interval> [] | <Literal Datetime> [] | { CURRENT | SYSDATE } [<DATETIME Field Qualifier> []] | { CURRENT_USER | DBSERVERNAME | SITENAME | TODAY | USER | NULL } } "

If you specify a default value, the global variable is initialized with the specified value.

CURRENT

CURRENT is a valid default only for a DATETIME variable. If the YEAR TO FRACTION(3) is its declared precision, no qualifier is needed. Otherwise, you must specify the same DATETIME qualifier when **CURRENT** is the default, as in the following example of a DATETIME variable:

```
DEFINE GLOBAL d_var DATETIME YEAR TO MONTH
      DEFAULT CURRENT YEAR TO MONTH;
```

SYSDATE

SYSDATE is a valid default only for a DATETIME variable. If the YEAR TO FRACTION(5) is the declared precision of the variable, no qualifier is needed. Otherwise, you must specify the same DATETIME qualifier when **SYSDATE** is the default, as in the following example of a DATETIME variable:

```
DEFINE GLOBAL dt_var DATETIME YEAR TO DAY
      DEFAULT SYSDATE YEAR TO DAY;
```

USER

If you use the value that **USER**, or its synonym **CURRENT_USER**, returns as the default, the variable must be defined as a CHAR, VARCHAR, NCHAR, or NVARCHAR data type. It is recommended that the length of the variable be at least 32 bytes. You risk getting an error message during INSERT and ALTER TABLE operations if the length of the variable is too small to store the default value.

TODAY

If you use **TODAY** as the default, the variable must be a DATE value. (See Constant Expressions on page 1040 for descriptions of **TODAY** and of the other system constants that can appear in the Default Value clause.)

BYTE and TEXT

The only default value valid for a BYTE or TEXT variable is NULL. The following example defines a TEXT global variable that is called **l_blob**:

```
CREATE PROCEDURE use_text()
   DEFINE i INT;
   DEFINE GLOBAL l_blob REFERENCES TEXT DEFAULT NULL;
   ...
END PROCEDURE
```

Here the REFERENCES keyword is required, because the DEFINE statement cannot declare a BYTE or TEXT data type directly; the **l_blob** variable is a pointer to a TEXT value that is stored in the global environment.

SITENAME or DBSERVERNAME

If you use the SITENAME or DBSERVERNAME keyword as the default, the variable must be a CHAR, VARCHAR, NCHAR, NVARCHAR, or LVARCHAR data type. Its default value is the name of the database server at runtime. It is recommended

that the size of the variable be at least 128 bytes long. You risk getting an error message during INSERT and ALTER TABLE operations if the length of the variable is too small to store the default value.

The following example uses the SITENAME keyword to specify a default value. This example also initializes a global BYTE variable to NULL:

```
CREATE PROCEDURE gl_def()
   DEFINE GLOBAL gl_site CHAR(200) DEFAULT SITENAME;
   DEFINE GLOBAL gl_byte REFERENCES BYTE DEFAULT NULL;
   ...
END PROCEDURE
```

Declaring Local Variables

A *local variable* has as its scope of reference the routine in which it is declared. If you omit the GLOBAL keyword, any variables declared in the DEFINE statement are local variables, and are not visible in other SPL routines.

For this reason, different SPL routines that declare local variables of the same name can run without conflict in the same DB-Access or SQL administration API session.

If a local variable and a global variable have the same name, the global variable is not visible within the SPL routine where the local variable is declared. (In all other SPL routines, only the global variable is in scope.)

The following DEFINE statement syntax is for declaring local variables:

(explicit id unique_1332_Connect_42_defs003) unique_1332_Connect_42_defs003 **DEFINE** *SPL_var* { *data_type* | **REFERENCES** { **BYTE** | **TEXT** } | **LIKE** { *view* | *synonym* | *table* } .*column* | { **PROCEDURE** } | { { **BLOB** | **CLOB** } | <Subset of Complex Data Types> | *distinct_type* | *opaque_type* } } ;

Element	Description	Restrictions	Syntax
column	Column name	Must already exist in the *table* or *view*	Identifier on page 1231;
data_type	Type of *SPL_var*	Cannot be BIGSERIAL, BYTE, SERIAL, SERIAL8, or TEXT	Data Type on page 985
distinct_type	A distinct type	Must already be defined in the database	Identifier on page 1231
opaque_type	An opaque type	Must already be defined in the database	Identifier on page 1231
SPL_var	New SPL variable	Must be unique within statement block	Identifier on page 1231;
synonym, *table*, *view*	Name of a table, view, or synonym	Synonym and the table or view to which it points must already exist when the statement is issued	Database Object Name on page 1225

Local variables do not support default values. The following example shows some typical definitions of local variables:

```
CREATE PROCEDURE def_ex()
   DEFINE i INT;
   DEFINE word CHAR(15);
   DEFINE b_day DATE;
   DEFINE c_name LIKE customer.fname;
   DEFINE b_text REFERENCES TEXT;
END PROCEDURE
```

Subset of Complex Data Types

To store one or more data values in a complex data type, the DEFINE statement can declare a local SPL variable as a collection of type SET, MULTISET, or LIST, with elements of a specified type, or as a generic COLLECTION. The DEFINE statement can also declare a local SPL variable as a named or an unnamed ROW type with an ordered set of fields, or as a generic ROW type.

Complex Data Types (Subset)

" { **COLLECTION** | { **SET** | **MULTISET** | **LIST** } ({ *data_type* | { **SET** | **MULTISET** | **LIST** } (*data_type* **NOT NULL**) } **NOT NULL**) | { *row* | **ROW** [(*fielddata_type*)] } } "

Element	Description	Restrictions	Syntax
data_type	Data type of collection elements, or of fields in an unnamed ROW type	Must match the data type of the values that the variable will store. Cannot be BIGSERIAL, BLOB, BYTE, CLOB, SERIAL, SERIAL8, or TEXT.	Data Type on page 985
field	Field of an unnamed ROW	If you specify no *field* list, the generic ROW variable can store data of any ROW type	Identifier on page 1231
row	Named ROW data type	Must exist in the database	Identifier on page 1231

Example of a named ROW type variable

The following statement registers in the database a named ROW data type that can store two non-NULL integer values, and declares **two_whole_numbers_t** as its identifier:

```
CREATE ROW TYPE IF NOT EXISTS
   two_whole_numbers_t( x INT NOT NULL, y INT NOT NULL);
```

Because this CREATE ROW TYPE statement specifies no supertype, the new named ROW type inherits no fields from any existing ROW type hierarchy.

This DEFINE statement declares **numbers** as an SPL variable whose data type is the **two_whole_numbers_t** named ROW type:

```
DEFINE numbers two_whole_numbers_t;   -- named ROW type
```

In the statement above, the name of this ROW type identifies the structure of the **numbers** variable as having two fields of type INT that can store only non-NULL values.

Example of a generic ROW type variable

The next example assigns to the variable **whatever** a generic ROW data type, without field declarations:

```
DEFINE whatever ROW;   -- generic ROW type
```

If an SPL function included both of these DEFINE statements, the **numbers** variable could return two integers as the result of the SPL routine, but the results cannot be returned to the calling context by the **whatever** variable.

Important:

Generic ROW and generic COLLECTION types are not supported as return data types of SPL functions. In the example above, the SPL routine must cast the fields of the generic ROW variable **whatever** into specific SQL data types.

Similarly, if elements of a generic COLLECTION variable store the routine results, those values must be cast into specific SQL data types that the SPL routine can return.

Declaring Collection Variables

A local variable of type COLLECTION, SET, MULTISET, or LIST can hold a collection of values fetched from the database. You cannot define a collection variable as global (with the GLOBAL keyword) or with a default value.

A variable declared with the keyword COLLECTION is an untyped (or generic) collection variable that can hold a collection of any data type.

A variable declared as type SET, MULTISET, or LIST is a *typed collection variable*. It can hold a collection of its specified data type only.

You must use the NOT NULL keywords when you define the elements of a typed collection variable, as in the following examples:

```
DEFINE a SET ( INT NOT NULL );

DEFINE b MULTISET ( ROW ( b1 INT,
                    b2 CHAR(50)
                  ) NOT NULL );

DEFINE c LIST( SET( INTEGER NOT NULL ) NOT NULL );
```

With variable **c**, both the INTEGER values in the SET and the SET values in the LIST are defined as NOT NULL.

You can define collection variables with nested complex types to hold matching nested complex type data. Any type or depth of nesting is allowed. You can nest **ROW** types within collection types, collection types within **ROW** types, collection types within collection types, **ROW** types within collection and **ROW** types, and so on.

If you declare a variable as COLLECTION type, the variable acquires varying data type declarations if it is reassigned within the same statement block, as in the following example:

```
DEFINE a COLLECTION;
LET a = setB;
```

```
...
LET a = listC;
```

In this example, **varA** is a generic collection variable that changes its data type to the data type of the currently assigned collection. The first LET statement makes **varA** a SET variable. The second LET statement makes **varA** a LIST variable.

Declaring ROW Variables

ROW variables hold data from named or unnamed ROW types. You can define a generic ROW variable, a named ROW variable, or an unnamed ROW variable.

A generic ROW variable, defined with the ROW keyword, can hold data from any ROW type. A named ROW variable holds data from the named ROW type that you specified in the declaration of the variable.

The following statements show examples of generic ROW variables and named ROW variables:

```
DEFINE d ROW;               -- generic ROW variable

DEFINE rectv rectangle_t;   -- named ROW variable
```

A named ROW variable holds named ROW types of the same type in the declaration of the variable.

To define a variable that will hold data stored in an unnamed ROW type, use the ROW keyword followed by the fields of the ROW type, as in:

```
DEFINE area ROW ( x int, y char(10) );
```

Unnamed ROW types are type-checked only by structural equivalence. Two unnamed ROW types are considered equivalent if they have the same number of fields, and if the fields have the same type definitions. Therefore, you could fetch either of the following ROW types into the variable **area** defined above:

```
ROW ( a int, b char(10) )
ROW ( area int, name char(10) )
```

ROW variables can have fields, just as ROW types have fields. To assign a value to a field of a ROW variable, use the qualifier notation *variableName.fieldName*, followed by an expression, as in the following example:

```
CREATE ROW TYPE rectangle_t (start point_t, length real, width real);

DEFINE r rectangle_t;
      -- Define a variable of a named ROW type
LET r.length = 45.5;
      -- Assign a value to a field of the variable
```

When you assign a value to a ROW variable, you can use any valid expression.

Declaring Opaque-Type Variables

Opaque-type variables hold data retrieved from opaque data types, which you create with the CREATE OPAQUE TYPE statement. An opaque-type variable can only hold data of the same opaque type on which it is defined. The following example defines a variable of the opaque type **point**, which holds the **x** and **y** coordinates of a two-dimensional point:

```
DEFINE b point;
```

Declaring Variables LIKE Columns

If you use the LIKE clause, the database server assigns the variable the same data type as a specified column in a table, synonym, or view.

The data types of variables that are defined as database columns are resolved at runtime; therefore, *column* and *table* do not need to exist at compile time.

You can use the LIKE keyword to declare that a variable is like a serial column. This declares:

- An INTEGER variable if the column is of the SERIAL data type
- An INT8 variable if the column is of the SERIAL8 data type
- A BIGINT variable if the column is of the BIGSERIAL data type

For example, if the column **serialcol** in the **mytab** table has the SERIAL data type, you can create the following SPL function:

```
CREATE FUNCTION func1()
DEFINE local_var LIKE mytab.serialcol;
RETURN;
END FUNCTION;
```

The variable **local_var** is treated as an INTEGER variable.

Defining Variables with Logical Character Semantics

When the SQL_LOGICAL_CHAR configuration parameter has specified for the current session) is set to '`ON`' or to a value greater than `1`, Informix® interprets size declarations as logical characters, rather than as bytes, in declarations of SPL variables of the following data types:

- CHAR or CHARACTER
- CHARACTER VARYING or VARCHAR
- LVARCHAR
- NCHAR
- NVARCHAR
- DISTINCT types whose base types are built-in character data types
- DISTINCT types whose base types are the previously listed data types
- ROW data type fields of any of the previously listed data types.
- Elements of these data types in LIST, MULTISET, or SET collection data types.

Enabling logical character semantics for the database locale guarantees that sufficient storage is available for the data type to store the specified number of logical characters. The resulting size in bytes of the SPL variable is the product of the declared size of the data type multiplied by the SQL_LOGICAL_CHAR value, if this is 2, 3, or 4, or (if SQL_LOGICAL_CHAR is set to '`ON`') by the number of bytes of storage that the largest logical character in the code set of the database locale requires.

If a client session connects to a database in which the SQL_LOGICAL_CHAR configuration parameter was enabled at the time of database creation, that setting takes effect at connection time.

DEFINE statements that use the LIKE keyword in datatype declarations create SPL variables whose data types match the schema of the column that the LIKE specification references. The SQL_LOGICAL_CHAR setting, if any is defined, has no effect on the size in memory of variables that DEFINE declares with the LIKE keyword.

For more information about the effect of the SQL_LOGICAL_CHAR setting in locales that use a multibyte code set, such as **UTF-8**, where a single logical character can require more than one byte of storage, see the description of the SQL_LOGICAL_CHAR configuration parameter in your *HCL® Informix® Administrator's Reference*. For additional information about multibyte locales and logical characters, see the *HCL® Informix® GLS User's Guide*.

Declaring Variables as the PROCEDURE Type

The PROCEDURE keyword indicates that in the current scope, the variable is a call to a UDR.

The DEFINE statement does not support a FUNCTION keyword. Use the PROCEDURE keyword, whether you are calling a user-defined procedure or a user-defined function.

Declaring a variable as PROCEDURE type indicates that in the current statement scope, the variable is not a call to a built-in function. For example, the following statement defines **length** as an SPL routine, not as the built-in LENGTH function:

```
DEFINE length PROCEDURE;
...
LET x = length (a,b,c)
```

This definition disables the built-in LENGTH function within the scope of the statement block. You would use such a definition if you had already created a user-defined routine with the name **length**.

If you create an SPL routine with the same name as an aggregate function (SUM, MAX, MIN, AVG, COUNT) or with the name **extend**, you must qualify the routine name with the owner name.

Declaring Variables for BYTE and TEXT Data

The keyword REFERENCES indicates that the variable does not contain a BYTE or TEXT value but is a pointer to the BYTE or TEXT value. Use the variable as though it holds the data.

The following example defines a local BYTE variable:

```
CREATE PROCEDURE use_byte()
   DEFINE i INT;
   DEFINE l_byte REFERENCES BYTE;
END PROCEDURE --use_byte
```

If you pass a variable of BYTE or TEXT data type to an SPL routine, the data is passed to the database server and stored in the root dbspace or dbspaces that the **DBSPACETEMP** environment variable specifies, if it is set. You do not need to know the location or name of the file that holds the data. BYTE or TEXT manipulation requires only the name of the BYTE or TEXT variable as it is defined in the routine.

EXIT

The EXIT statement can terminate FOR, FOREACH, LOOP, or WHILE statements.

Syntax

```
EXIT { FOREACH | [ { FOR | LOOP | WHILE } ] [ [ label ] WHEN condition ] } ;
```

Element	Description	Restrictions	Syntax
condition	Loop terminates when this evaluates to TRUE.	If *condition* evaluates to FALSE, the loop continues.	Condition on page 967
label	Label of a loop from which to exit	Must be the label of a loop statement that includes the EXIT statement	Identifier on page 1231

Usage

The EXIT statement transfers control of execution from an iterative statement, causing the innermost loop of the enclosing statement type (FOR, FOREACH, LOOP, or WHILE) to terminate. If no loop label or WHEN condition is specified, execution resumes at the first statement that follows the current FOR, FOREACH, LOOP, or WHILE statement.

EXIT From FOREACH Statements

If the EXIT statement has the FOREACH statement as its innermost enclosing statement, the FOREACH keyword must immediately follow the EXIT keyword. The EXIT FOREACH statement unconditionally terminates the FOREACH statement, or else returns an error, if no FOREACH statement encloses the EXIT FOREACH statement.

The following program fragment includes the EXIT FOREACH statement:

```
FOREACH cursor1 FOR
      SELECT * INTO a FROM TABLE(b);
      IF a = 4 THEN
         DELETE FROM TABLE(b)
            WHERE CURRENT OF cursor1;4
         EXIT FOREACH;
      END IF;
   END FOREACH;
```

EXIT From FOR, LOOP, and WHILE Loops

If the EXIT statement is issued outside the FOREACH statement, it returns an error unless it is issued from the FOR, FOR LOOP, LOOP, WHILE LOOP, or WHILE statement as its innermost enclosing statement. In FOR or WHILE statements that do not include the LOOP keyword, the corresponding FOR or WHILE keyword is required after the EXIT keyword. Execution resumes at the first executable statement that follows the innermost loop from which the EXIT statement was issued.

The EXIT statement requires no other keyword when it is issued from the FOR LOOP, LOOP, or WHILE LOOP statement, with or without a loop label, but if you include the FOR, LOOP, or WHILE keyword after the EXIT keyword, that keyword must correspond to the type of loop from which the EXIT statement is issued.

If the EXIT keyword is followed by the identifier of a loop label, and no *condition* is specified, execution resumes at the first executable statement that follows the FOR, FOR LOOP, LOOP, WHILE LOOP, or WHILE statement whose label is specified. This enables the EXIT statement to exit from nested loops, if an outer loop is labeled.

If a WHEN *condition* follows the EXIT or EXIT *label* specification, EXIT has no effect unless the *condition* is true. If the condition is true, execution resumes after the labeled loop, or after the innermost loop, if no *label* is specified.

If the database server cannot find the specified loop or loop label, the EXIT statement fails. If EXIT is issued outside any FOR, FOREACH, LOOP, or WHILE statement, it generates errors.

The following example uses an EXIT FOR statement. In the FOR loop, when `j` becomes `6`, the IF condition `i = 5` in the WHILE loop is true. The FOR loop stops executing, and the SPL procedure continues at the next statement outside the FOR loop (in this case, the END PROCEDURE statement). In this example, the procedure ends when `j` equals `6`:

```
CREATE PROCEDURE ex_cont_ex()
   DEFINE i,s,j, INT;
   FOR j = 1 TO 20
      IF j > 10 THEN
         CONTINUE FOR;
      END IF
      LET i,s = j,0;
      WHILE i > 0
         LET i = i -1;
         IF i = 5 THEN
            EXIT FOR;
         END IF
      END WHILE
   END FOR
END PROCEDURE;
```

The following program fragment shows two conditional EXIT statements in a labeled WHILE LOOP statement that is nested within another labeled LOOP statement:

```
        <<outer>>
        LOOP
        LET x = x+1;
           <<inner>>
           WHILE ( i >10 ) LOOP
              LET x = x+1;
              EXIT inner WHEN x = 2;
              EXIT outer WHEN x > 3;
              END LOOP inner;
        LET x = x+1;
        END LOOP outer;
```

When the `x=2` condition is true, the `EXIT inner` statement transfers control to the LET statement that follows the loop whose label is **inner**. When the `x>3` condition is true, the `EXIT outer` statement terminates execution of the **outer** loop.

Related Statements

<< Label >> statement on page 913, FOR on page 931, FOREACH on page 935, LOOP on page 945, WHILE on page 961

FOR

Use the FOR statement to initiate a controlled (definite) loop when you want to guarantee termination of the loop. The FOR statement uses expressions or range operators to specify a finite number of iterations for a loop.

Syntax

(explicit id unique_1309_Connect_42_for002) unique_1309_Connect_42_for002(explicit id unique_1309_Connect_42_for003) unique_1309_Connect_42_for003

[<<*label* >>] **FOR***loop_var* { **IN**({ < Range> | *expression* }) | = < Range> } { <Statement Block>[] **END FOR** [*label*[]] | **LOOP** <Statement Block>[] **END LOOP** [*label*[]] } [;]

Range

" *left_expr***TO***right_expr* "

" [**STEP***increment_expr*] "

Element	Description	Restrictions	Syntax
expression	Value to compare with *loop_var*	Must match *loop_var* data type	Expression on page 1003
increment_expr	Positive or negative value by which *loop_var* is incremented. Default is either 1 (if *left_expr* < *right_expr*), or else -1 (if *left_expr* > *right_expr*).	Must return an integer. Cannot return `0`.	Expression on page 1003
label	Name of the loop label for this loop	Must exist and must be unique among label names in this SPL routine	Identifier on page 1231
left_expr	Starting expression of a range	Value must match SMALLINT or INT data type of *loop_var*, but *left_expr* must not equal *right_expr*	Expression on page 1003
loop_var	Variable that determines how many times the loop executes	Must be defined and in scope within this statement block	Identifier on page 1231
right_expr	Ending expression in the range	Same as for *left_expr*	Expression on page 1003

Usage

The database server evaluates all expressions before the FOR statement executes. If one or more of the expressions are variables whose values change during the loop, the change has no effect on the iterations of the loop.

You can use the output from a SELECT statement as the *expression*.

The FOR loop terminates when *loop_var* is equal to the values of each element in the expression list or range in succession, or when it encounters an EXIT FOR statement. An error is issued, however, if an assignment within the body of the FOR statement attempts to modify the value of *loop_var*.

The size of *right_expr* relative to *left_expr* determine whether the range is stepped through by positive or by negative increments:

- The increments are positive if *left_expr* < *right_expr*.
- The increments are negative if *left_expr* > *right_expr*.

If you specify no *increment_expr*, the default size of each step is 1, with a positive or negative sign determined by the rules above.

Using the TO Keyword to Define a Range

The TO keyword implies a range operator. The range is defined by *left_expression* and *right_expression*, and the STEP *increment_expr* option implicitly sets the number of increments. If you use the TO keyword, *loop_var* must be an INT or SMALLINT data type.

The next example shows two equivalent FOR statements. Each uses the TO keyword to define a range. The first uses the IN keyword, and the second uses an equal sign (=). Each statement causes the loop to execute five times:

```
FOR index_var IN (12 TO 21 STEP 2)
   -- statement block
END FOR;

FOR index_var = 12 TO 21 STEP 2
   -- statement block
END FOR;
```

If you omit the STEP option, the database server gives *increment_expr* the value of `-1` if *right_expression* is less than *left_expression*, or `+1` if *right_expression* is more than *left_expression*. If *increment_expr* is specified, it must be negative if *right_expression* is less than *left_expression*, or positive if *right expression* is more than *left_expression*.

The two statements in the following example are equivalent. In the first statement, the STEP increment is explicit. In the second statement, the STEP increment is implicitly `1`:

```
FOR index IN (12 TO 21 STEP 1)
   -- statement block
END FOR;

FOR index = 12 TO 21
   -- statement block
END FOR;
```

The database server initializes the value of *loop_var* to the value of *left_expression*. In subsequent iterations, the server adds *increment_expr* to the value of *loop_var* and checks *increment_expr* to determine whether the value of *loop_var* is still between *left_expression* and *right_expression*. If so, the next iteration occurs. Otherwise, an exit from the loop takes place. Or, if you specify another range, the variable takes on the value of the first element in the next range.

Specifying Two or More Ranges in a Single FOR Statement

The following example shows a statement that traverses a loop forward and backward and uses different increment values for each direction:

```
FOR index_var IN (15 to 21 STEP 2, 21 to 15 STEP -3)
   -- statement body
END FOR;
```

Using an Expression List as the Range

The database server initializes the value of *loop_var* to the value of the first expression specified. In subsequent iterations, *loop_var* takes on the value of the next expression. When the database server has evaluated the last expression in the list and used it, the loop stops.

The expressions in the IN list do not need to be numeric values, as long as you do not use range operators in the IN list. The following example uses a character expression list:

```
FOR c IN ('hello', (SELECT name FROM t), 'world', v1, v2)
   INSERT INTO t VALUES (c);
END FOR;
```

The following FOR statement shows the use of a numeric expression list:

```
FOR index IN (15,16,17,18,19,20,21)
   -- statement block
END FOR;
```

Mixing Range and Expression Lists in the Same FOR Statement

If *loop_var* is an INT or SMALLINT value, you can mix ranges and expression lists in the same FOR statement. The following example shows a mixture that uses an integer variable. Values in the expression list include the value that is returned from a SELECT statement, a sum of an integer variable and a constant, the values that are returned from an SPL function named **p_get_int**, and integer constants:

```
CREATE PROCEDURE for_ex ()
   DEFINE i, j INT;
   LET j = 10;
   FOR i IN (1 TO 20, (SELECT c1 FROM tab WHERE id = 1),
         j+20 to j-20, p_get_int(99),98,90 to 80 step -2)
      INSERT INTO tab VALUES (i);
   END FOR;
END PROCEDURE;
```

Specifying a Labelled FOR Loop

About this task

To create a labeled FOR loop, declare a loop label before the initial FOR keyword, and repeat the label after the END FOR keywords, as in this example:

```
CREATE PROCEDURE ex_cont_ex()
   DEFINE i,s,j, INT;
   <<for_lab>>
   FOR j = 1 TO 20
      IF j > 10 THEN
         CONTINUE FOR;
      END IF
      LET i,s = j,0;
```

```
      WHILE i > 0
         LET i = i -1;
         IF i = 5 THEN
            EXIT for_lab;
         END IF
      END WHILE
   END FOR for_lab
END PROCEDURE;
```

Here the `EXIT for_lab` statement has the same effect that the `EXIT` or `EXIT FOR` keywords would have, terminating both the FOR loop and the routine. In this example, the statement that includes the `EXIT for_lab` statement has the same effect that `EXIT for_lab WHEN i = 5` would have.

You can also label a LOOP statement that begins with a loop <<*label*>> specification that immediately precedes the initial FOR keyword. In this type of loop, the CONTINUE LOOP, EXIT LOOP, and END LOOP keywords replace the CONTINUE FOR, EXIT FOR, and END FOR keywords. Both the LOOP and FOR keywords are optional after the CONTINUE and EXIT keywords, but the END LOOP keywords are required in SPL loop statements that include the LOOP keyword.

You can use similar syntax to create an unlabeled loop that omits the <<*label*>> specification that immediately precedes the initial FOR keyword. In this case, you must also omit the undelimited loop label identifier that follows the END LOOP keywords. See the LOOP statement for a description and examples of these forms of labeled and unlabeled loop statements that enable you to combine FOR statement syntax, with a finite number of loop iterations, with the "loop forever" syntax of the LOOP statement.

Example

Related Statements

<< Label >> statement on page 913, CONTINUE on page 919, EXIT on page 929, LOOP on page 945, FOREACH on page 935, WHILE on page 961

FOREACH

Use the FOREACH statement to declare a direct cursor that can select and manipulate more than one row from a the result set of a query, or more than one element from a collection.

Syntax

Direct sequential cursors that the FOREACH statement of SPL can create are distinct from the dynamic cursors that the DECLARE statement of SQL can create in SPL routines. (For the syntax and usage of dynamic cursors in SPL routines, see Declaring a Dynamic Cursor in an SPL Routine on page 449.)

(explicit id unique_617_Connect_42_frch003) unique_617_Connect_42_frch003(explicit id unique_617_Connect_42_frch004)
unique_617_Connect_42_frch004(explicit id unique_617_Connect_42_frch005) unique_617_Connect_42_frch005 **FOREACH** { [{ **WITH HOLD** | *cursor* [**WITH HOLD**] **FOR** | <Routine Call> [**INTO** *data_var*] }] <SELECT ... INTO Statement>[] } <Statement Block>[] **END FOREACH** [;]

Routine Call

" **EXECUTE** { **PROCEDURE** { *procedure* | *SPL_var* | *function* } | **FUNCTION** { *SPL_var* | *function* } } "

" ([<Argument>[]]) "

Element	Description	Restrictions	Syntax
cursor	Identifier that you declare here as the name of this direct cursor	Must be unique among names of cursors, prepared statements, and SPL variables in the routine	Identifier on page 1231
data_var	SPL variable in the calling routine that receives the returned values	Data type of *data_var* must be appropriate for returned value	Identifier on page 1231
function, *procedure*	SPL function or procedure to execute	Function or procedure must exist	Database Object Name on page 1225
SPL_var	SPL variable that contains the name of a routine to execute	Must be of type CHAR, VARCHAR, NCHAR, or NVARCHAR	Identifier on page 1231

Usage

To execute a FOREACH statement, the database server takes these actions:

1. It declares and implicitly opens a direct sequential cursor.
2. It obtains the first row from the query contained within the FOREACH loop, or else the first set of values from the called routine.
3. It assigns to each variable in the variable list the value of the corresponding value from the active set that the SELECT statement or the called routine creates.
4. It executes the statement block.
5. It fetches the next row from the SELECT statement or called routine on each iteration, and it repeats steps 3 and 4.
6. It terminates the loop when it finds no more rows that satisfy the SELECT statement or called routine. It closes the direct sequential cursor when the loop terminates.

Because the statement block can contain additional FOREACH statements, cursors can be nested. No limit exists on the number of nested cursors.

An SPL routine that returns more than one row, collection element, or set of values is called a *cursor function*. An SPL routine that returns only one row or value is called a *noncursor function*.

This SPL procedure illustrates FOREACH statements with a SELECT ... INTO clause, with an explicitly named cursor, and with a procedure call:

```
CREATE PROCEDURE foreach_ex()
   DEFINE i, j INT;
   FOREACH SELECT c1 INTO i FROM tab ORDER BY 1
      INSERT INTO tab2 VALUES (i);
   END FOREACH
   FOREACH cur1 FOR SELECT c2, c3 INTO i, j FROM tab
      IF j > 100 THEN
         DELETE FROM tab WHERE CURRENT OF cur1;
         CONTINUE FOREACH;
      END IF
      UPDATE tab SET c2 = c2 + 10 WHERE CURRENT OF cur1;
   END FOREACH
   FOREACH EXECUTE PROCEDURE bar(10,20) INTO i
      INSERT INTO tab2 VALUES (i);
   END FOREACH
END PROCEDURE; -- foreach_ex
```

A Select cursor is closed when any of the following situations occur:

- The cursor returns no further rows.
- The cursor is a Select cursor without a HOLD specification, and a transaction completes using the COMMIT or ROLLBACK statement.
- An EXIT statement executes, which transfers control out of the FOREACH statement.
- An exception occurs that is not trapped inside the body of the FOREACH statement. (See ON EXCEPTION on page 949.)
- A cursor in the calling routine that is executing this cursor routine (within a FOREACH loop) closes for any reason.

Note:

The FOREACH statement cannot define a SCROLL cursor. Each FOREACH cursor is a sequential cursor, which can fetch only the next row in sequence from the active set. A cursor that FOREACH defines can read through the active set only once each time it is opened.

Using a SELECT ... INTO Statement

As indicated in the diagram for FOREACH on page 935, not all clauses and options of the SELECT statement are available for you to use in a FOREACH statement. The SELECT statement in the FOREACH statement must include the INTO clause. It can also include UNION and ORDER BY clauses, but it cannot use the INTO TEMP clause. For a complete description of SELECT syntax and usage, see SELECT statement on page 690. The data type and count of each variable in the variable list must match each value that the `SELECT ... INTO` statement returns.

The database server issues an error if you include a semicolon (`;`) within the FOREACH statement to terminate the `SELECT ... INTO` specification. The following program fragment, for example, fails with a syntax error:

```
CREATE DBA PROCEDURE IF NOT EXISTS shapes()
```

```
   DEFINE vertexes SET( point NOT NULL );
   DEFINE pnt point;

   SELECT definition INTO vertexes FROM polygons
      WHERE id = 207;

   FOREACH cursor1 FOR
      SELECT * INTO pnt FROM TABLE(vertexes); -- Semicolon not valid
   . . .
   END FOREACH
. . .
END PROCEDURE;
```

In the example above, you can avoid this error by deleting the semicolon that immediately follows the `TABLE(vertexes)` specification.

Using the ORDER BY Clause of the SELECT Statement

The ORDER BY clause of the SELECT statement implies that the query returns more than one row. Unless you use the DECLARE statement of SQL to define a Select cursor or a Function cursor, the database server issues an error if you specify the ORDER BY clause outside the context of a FOREACH loop to process the returned rows individually within an SPL routine.

For the syntax and usage of the DECLARE statement in SPL routines, see Declaring a Dynamic Cursor in an SPL Routine on page 449.

Using Hold Cursors

The WITH HOLD keywords specify that the cursor should remain open when a transaction closes (by being committed or by being rolled back).

Updating or Deleting Rows Identified by Cursor Name

Specify a *cursor* name in the FOREACH statement if you intend to use the WHERE CURRENT OF *cursor* clause in UPDATE or DELETE statements that operate on the current row of *cursor* within the FOREACH loop. Although you cannot include the FOR UPDATE keywords in the SELECT ... INTO segment of the FOREACH statement, the cursor behaves like a FOR UPDATE cursor.

For a discussion of locking, see the section on Locking with an Update Cursor on page 440. For a discussion of isolation levels, see the description of SET ISOLATION statement on page 812.

Using Collection Variables

The FOREACH statement allows you to declare a cursor for an SPL collection variable. Such a cursor is called a *Collection cursor*. Use a collection variable to access the elements of a collection (SET, MULTISET, LIST) column. Use a cursor when you want to access one or more elements in a collection variable.

The following excerpt from an SPL routine shows how to fill a collection variable and then how to use a cursor to access individual elements:

```
DEFINE a SMALLINT;
DEFINE b SET(SMALLINT NOT NULL);
SELECT numbers INTO b FROM table1 WHERE id = 207;
FOREACH cursor1 FOR
   SELECT * INTO a FROM TABLE(b);
...
END FOREACH;
```

In this example, the SELECT statement selects one element at a time from the collection variable **b** into the element variable **a**. The projection list is an asterisk, because the collection variable **b** contains a collection of built-in types. The variable **b** is used with the TABLE keyword as a Collection-Derived Table. For more information, see Collection-Derived Table on page 1212.

The next example also shows how to fill a collection variable and then how to use a cursor to access individual elements. This example, however, uses a list of ROW-type fields in its projection list:

```
DEFINE employees employee_t;
DEFINE n VARCHAR(30);
DEFINE s INTEGER;

SELECT emp_list into employees FROM dept_table
   WHERE dept_no = 1057;
FOREACH cursor1 FOR
   SELECT name,salary
      INTO n,s FROM TABLE( employees ) AS e;
...
END FOREACH;
```

Here the collection variable **employees** contains a collection of ROW types. Each ROW type contains the fields **name** and **salary**. The collection query selects one name and salary combination at a time, placing **name** into **n** and **salary** into **s**. The AS keyword declares **e** as an alias for the collection-derived table **employees**. The alias exists as long as the SELECT statement executes.

Modifying Elements in a Collection Variable

To update an element of a collection within an SPL routine, you must first declare a cursor with the FOREACH statement.

Then, within the FOREACH loop, select elements one at a time from the collection variable, using the collection variable as a collection-derived table in a SELECT query.

When the cursor is positioned on the element to be updated, you can use the WHERE CURRENT OF clause, as follows:

- The UPDATE statement with the WHERE CURRENT OF clause updates the value in the current element of the collection variable.
- The DELETE statement with the WHERE CURRENT OF clause deletes the current element from the collection variable.

Using Select Cursors with FOREACH

When using the FOREACH statement, if the result set from a query is to be modified, do not use this result set as an exit criterion for the FOREACH loop. For example, if the FOREACH statement declares a Select cursor that is expected to return

30 rows, but DELETE, INSERT, or UPDATE operations within the FOREACH loop modify the result set of the query, this might cause unexpected behavior. To ensure that a FOREACH loop works as intended, make sure that any Select cursor in the FOREACH statement completes its execution before you begin modifying its result set.

One way to avoid unexpected results from a FOREACH loop that performs DML operations on the rows returned by a query is to use an ORDER BY clause in the SELECT statement to materialize the result set.

Calling a UDR in the FOREACH Loop

In general, use these guidelines for calling another UDR from an SPL routine:

- To call a user-defined procedure, use EXECUTE PROCEDURE *procedure name*.
- To call a user-defined function, use EXECUTE FUNCTION *function name* (or EXECUTE PROCEDURE *function name* if the user-defined function was created with the CREATE PROCEDURE statement).

If you use EXECUTE PROCEDURE, the database server looks first for a user-defined procedure of the name you specify. If it finds the procedure, the database server executes it. If it does not find the procedure, it looks for a user-defined function of the same name to execute. If the database server finds neither a function nor a procedure, it issues an error message. If you use EXECUTE FUNCTION, the database server looks for a user-defined function of the name you specify. If it does not find a function of that name, the database server issues an error message.

An SPL function can return zero (0) or more values or rows.

The data type and count of each variable in the variable list must match each value that the function returns.

Related Statements

CONTINUE on page 919, EXIT on page 929, FOR on page 931, LOOP on page 945,WHILE on page 961

GOTO

Use the GOTO statement to transfer control of program execution to the statement that has a specified statement label.

Syntax

```
GOTO label ;
```

Element	Description	Restrictions	Syntax
label	Name of the loop label for this loop	Must be unique among labels in this SPL routine	Identifier on page 1231

Usage

The GOTO statement branches to a statement label unconditionally. The statement label must be unique within its scope and must precede an executable statement. When successfully executed, the GOTO statement transfers control to the labeled statement or statement block.

In the following program fragment, the **jump_back** function transfers control to a LET statement that has the statement label **back** if the value of variable **j** is greater than 100.

```
CREATE FUNCTION jump_back()
   RETURNING INT;
   DEFINE i,j INT;
   ...
   <<back>>
   LET j = j + i
   FOR i IN (1 TO 52 STEP 5)
      IF i < 11 THEN
         LET j = j + 3
         CONTINUE FOR;
      END IF;
      IF j > 100 THEN
         GOTO back
      END IF;
      RETURN j WITH RESUME;
   END FOR;
END FUNCTION;
```

The GOTO statement is not valid in an ON EXCEPTION statement block.

The identifier of the statement label that the GOTO statement references must exist in the database, must be unique among statement labels and loop labels the SPL routine, and must be within a scope that the GOTO statement can reach.

Example

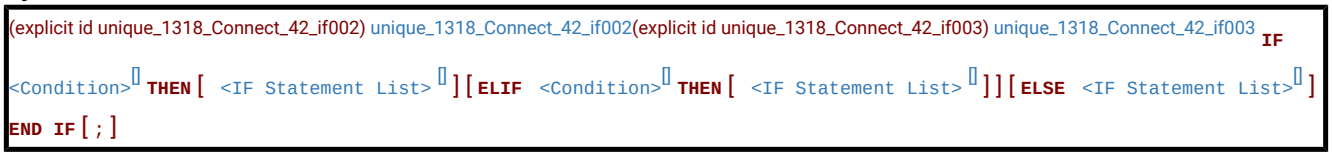
Related Statements

[<< Label >> statement on page 913]

IF

Use the IF statement to create a logical branch within an SPL routine.

Syntax



Usage

The database server processes the IF statement by the following steps:

1. If the condition that follows the IF keyword is true, any statements that follow the first THEN keyword of the IF statement execute, and the IF statement terminates.
2. If the result of the initial IF condition is false, but an ELIF clause exists, the database server evaluates the condition that follows the ELIF keyword.

3. If the result of the ELIF condition is true, any statements that follow the THEN keyword of the ELIF clause execute, and the IF statement terminates.
4. If the result of the condition in the first ELIF clause is also false, but one or more additional ELIF clauses exist, the database server evaluates the condition in the next ELIF clause, and proceeds as in the previous step if it is true. If it is false, the database server evaluates the condition in successive ELIF clauses, until it finds a condition that is true, in which case it executes the statement list that follows the THEN keyword of that ELIF clause, and the IF statement terminates.
5. If no condition in the IF statement is true, but the ELSE clause exists, statements that follow the ELSE keyword execute, and the IF statement terminates.
6. If none of the conditions in the IF statement are true, and no ELSE clause exists, the IF statement terminates without executing any statement list.

ELIF Clause

Use the ELIF clause to specify one or more additional conditions to evaluate. If the IF condition is false, the ELIF condition is evaluated. If the ELIF condition is true, the statements that follow the THEN keyword in the ELIF clause execute.

If no statement follows the THEN keyword of the ELIF clause when the ELIF condition is true, program control passes from the IF statement to the next statement.

ELSE Clause

The ELSE clause executes if no true previous condition exists in the IF clause or any of the ELIF clauses.

In the following example, the SPL function uses an IF statement with both an ELIF clause and an ELSE clause. The IF statement compares two strings.

The function displays `1` to indicate that the first string comes before the second string alphabetically, or `-1` if the first string comes after the second string alphabetically. If the strings are the same, a zero (0) is returned.

```
CREATE FUNCTION str_compare (str1 CHAR(20), str2 CHAR(20))
   RETURNING INT;
   DEFINE result INT;
      IF str1 > str2 THEN LET result =1;
         ELIF str2 > str1 THEN LET result = -1;
         ELSE LET result = 0;
      END IF
   RETURN result;
END FUNCTION -- str_compare
```

Conditions in an IF Statement

Just as in the WHILE statement, if any expression in the *condition* evaluates to NULL, then the condition cannot be true, unless you are explicitly testing for NULL using the IS NULL operator. The following rules summarize NULL values in conditions:

1. If the expression **x** evaluates to NULL, then **x** is not true by definition. Furthermore, NOT (**x**) is also not true .
2. IS NULL is the only operator that can return true for **x**. That is, **x** IS NULL is true, and **x** IS NOT NULL is not true.

If an expression in the condition has an UNKNOWN value from an uninitialized SPL variable, the statement terminates and raises an exception.

You can specify a trigger-type Boolean operator (DELETING, INSERTING, SELECTING, or UPDATING) as a condition in an IF statement only within a trigger routine.

IF Statement List

(explicit id unique_1354_Connect_42_ifsl002) unique_1354_Connect_42_ifsl002(explicit id unique_1354_Connect_42_ifsl003)

unique_1354_Connect_42_ifsl003(explicit id unique_1354_Connect_42_ifsl004) unique_1354_Connect_42_ifsl004

IF Statement List



LET

Use the LET statement to assign values to variables or to call a user-defined SPL routine and assign the returned value or values to SPL variables.

Syntax

(explicit id unique_1359_Connect_42_let002) unique_1359_Connect_42_let002(explicit id unique_1359_Connect_42_let003)

unique_1359_Connect_42_let003(explicit id unique_1359_Connect_42_let004) unique_1359_Connect_42_let004



Element	Description	Restrictions	Syntax
function	SPL function to be invoked	Must exist in the database	Identifier on page 1231
SPL_var	SPL variable to receive a value that the *function*, expression, or query returns	Must be defined and in scope within the statement block	Identifier on page 1231;

Usage

The LET statement can assign a value returned by an expression, function, or query to an SPL variable. At runtime, the value to be assigned is calculated first. The resulting value is cast to the data type of *SPL_var*, if possible, and the assignment occurs. If conversion is not possible, an error occurs, and the value of the variable remains undefined. (A LET operation that assigns a single value to a single SPL variable is called a *simple assignment*.)

A *compound assignment* assigns multiple expressions to multiple SPL variables. The data types of expressions in the expression list do not need to match the data types of the corresponding variables in the variable list, because the database server automatically converts the data types. (For a detailed discussion of casting, see the *HCL® Informix® Guide to SQL: Reference*.)

In multiple-assignment operations, the number of variables to the left of the equal (=) sign must match the number of values returned by the functions, expressions, and queries listed on the right of the equal (=) sign. The following example shows several LET statements that assign values to SPL variables:

```
LET a   = c + d ;
LET a,b = c,d ;
LET expire_dt = end_dt + 7 UNITS DAY;
LET name = 'Brunhilda';
LET sname = DBSERVERNAME;
LET this_day = TODAY;
```

You cannot use multiple values to the right of the equal (=) sign to operate on other values. For example, the following statement is not valid:

```
LET a,b = (c,d) + (10,15); -- INVALID EXPRESSION
```

Using a SELECT Statement in a LET Statement

The examples in this section use a SELECT statement in a LET statement. You can use a SELECT statement to assign values to one or more variables on the left side of the equals (=) operator, as the following example shows:

```
LET a,b = (SELECT c1,c2 FROM t WHERE id = 1);
LET a,b,c = (SELECT c1,c2 FROM t WHERE id = 1), 15;
```

You cannot use a SELECT statement to make multiple values operate on other values. The following example is invalid:

```
LET a,b = (SELECT c1,c2 FROM t) + (10,15); -- INVALID CODE
```

Because a LET statement is equivalent to a SELECT ... INTO statement, the two statements in the following example have the same results: `a=c` and `b=d`:

```
CREATE PROCEDURE proof()
   DEFINE a, b, c, d INT;
   LET a,b = (SELECT c1,c2 FROM t WHERE id = 1);
   SELECT c1, c2 INTO c, d FROM t WHERE id = 1
END PROCEDURE
```

If the SELECT statement returns more than one row, you must enclose the SELECT statement in a FOREACH loop.

For a description of SELECT syntax and usage, see SELECT statement on page 690.

Calling a Function in a LET Statement

You can call a user-defined function in a LET statement and assign the returned values to an SPL variable that receives the values that the function returns.

An SPL function can return multiple values (that is, values from multiple columns in the same row) into a list of variable names. In other words, the function can have multiple values in its RETURN statement and the LET statement can have multiple variables to receive the returned values.

When you call the function, you must specify all the necessary arguments to the function unless the arguments of the function have default values. If you specify the name of one of the parameters in the called function with syntax such as **name = 'smith'**, you must name all of the parameters.

An SPL function that selects and returns more than one row must be enclosed in a FOREACH loop.

The following two examples show valid LET statements:

```
LET a, b, c = func1(name = 'grok', age = 17);
LET a, b, c = 7, func2('orange', 'green');
```

The following LET statement is not valid because it tries to add the output of two functions and then assign the sum to two variables, **a** and **b**.

```
LET a, b = func1() + func2(); -- INVALID CODE
```

You can easily split this LET statement into two valid LET statements:

```
LET a = (func1() + func2());
LET b = a;                   -- VALID CODE
```

A function called in a LET statement can have an argument of COLLECTION, SET, MULTISET, or LIST. You can assign the value that the function returns to a variable, for example:

```
LET d = function1(collection1);
LET a = function2(set1);
```

In the first statement, the SPL function **function1** accepts **collection1** (that is, any collection data type) as an argument and returns its value to the variable **d**. In the second statement, the SPL function **function2** accepts **set1** as an argument and returns a value to the variable **a**.

LOOP

Use the LOOP statement to define a loop with an indeterminate number of iterations.

Syntax

(explicit id unique_1311_Connect_42_loop001) unique_1311_Connect_42_loop001(explicit id unique_1311_Connect_42_loop002)

unique_1311_Connect_42_loop002(explicit id unique_1311_Connect_42_loop003) unique_1311_Connect_42_loop003 [<<*label* >>] [{ **WHILE** <Condition> | **FOR***loop_var* { **IN** ({ < Range> | *expression* }) | = < Range> } }] **LOOP** <Statement Block> **END LOOP** [*label*] [;]

Range

" *left_expression***TO***right_expression* "

" [**STEP***increment_expr*] "

Element	Description	Restrictions	Syntax
expression	Value to compare with *loop_var*	Must match *loop_var* data type	Expression on page 1003

Element	Description	Restrictions	Syntax
increment_expr	Positive or negative value by which *loop_var* is incremented	Must return an integer. Cannot return `0`.	Expression on page 1003
label	Name of the loop label for this loop	Must be unique among labels in this SPL routine	Identifier on page 1231
left_expression	Starting expression of a range	Value must match SMALLINT or INT data type of *loop_var*	Expression on page 1003
loop_var	Variable that determines how many times the loop executes	Must be defined and in scope within this statement block	Identifier on page 1231
right_expression	Ending expression in the range	Same as for *left_expression*	Expression on page 1003

Usage

The LOOP statement is an iterative statement that resembles the FOR and WHILE statements. Like FOR and WHILE, the LOOP statement can have an optional loop label, It can include the CONTINUE statement to specify another iteration, and the EXIT statement to terminate execution of the loop.

Besides resembling FOR and WHILE in its functionality, the LOOP statement can use the syntax of FOR or WHILE that precedes the statement block. Sections that follow describe several forms of the LOOP statement, including these:

- Simple LOOP statements that iterate a statement loop indefinitely
- FOR LOOP statements, that use FOR statement syntax specify a finite number of iterations
- WHILE LOOP statements, that iterate while a specified condition is true
- Labeled versions of each of these LOOP statements, which can terminate deeply nested loops.

Simple LOOP Statements

The following program fragment illustrates a simple form of the LOOP statement.

```
LOOP
LET i = i + 1;
    IF i = 5 THEN EXIT;
    ELSE
    CONTINUE;
    END IF
END LOOP;
```

In this example the IF statement limits the number of iterations. Here the CONTINUE and EXIT statements omit the optional LOOP keyword, but the END LOOP statement is required at the end of the statement loop. A similar FOR or WHILE keyword would have required the FOR or WHILE keywords, respectively, in the CONTINUE and EXIT statements.

The next example uses a conditional EXIT statement to terminate the loop:

```
LOOP
LET i = i + 1;
   EXIT WHEN i = 4;
END LOOP;
```

No keyword identifying the type of loop statement is required after the EXIT statement, as would be the case for an EXIT statement in a FOR, WHILE, or FOREACH statement. When the `i = 4` condition becomes true, program control passes from the LOOP statement to whatever statement follows the END LOOP keywords.

FOR LOOP Statements

The FOR LOOP statement uses FOR statement syntax to specify a variable and a range of values that the variable can take. The loop iterates until the specified limit to these values is reached, or until control is transferred outside the loop, as by the unconditional EXIT statement in the following example:

```
FOR i IN (1 TO 5) LOOP
   IF i = 5 THEN EXIT;
   ELSE
   CONTINUE;
END LOOP;
```

In the FOR LOOP statement, the FOR keyword can follow the EXIT or CONTINUE keyword, but the FOR keyword is not required, as it is in an ordinary FOR statement.

The following example replaces the IF statement with a functionally equivalent conditional EXIT statement:

```
FOR i IN (1 TO 5) LOOP
   EXIT WHEN i = 5;
   END LOOP;
```

WHILE LOOP Statements

To create a WHILE LOOP statement, loop, you can immediately follow a WHILE *condition* specification with a LOOP statement. The resulting loop terminates after the *condition* becomes false, or when some other statement transfers program control from the loop. In the following WHILE LOOP statement, the condition specifies that the loop terminates after the loop variable **i** has been incremented to the value of 6:

```
WHILE (i < 6) LOOP
   LET i = i + 1;
   IF i = 5 THEN EXIT;
   ELSE
   CONTINUE;
   END IF
END LOOP;
```

As in the FOR LOOP statement, the EXIT and CONTINUE keywords do not need to specify the type of loop statement, but the example would not be affected if EXIT WHILE and CONTINUE WHILE replaced the EXIT and CONTINUE keywords. The END LOOP keywords are required, however, because Informix® treats the WHILE LOOP (and FOR LOOP) statements as LOOP statements, despite their initial FOR and WHILE specifications.

Labeled LOOP Statements

About this task

All forms of the LOOP statement, including the FOR LOOP, WHILE LOOP, and simple LOOP statements can have statement labels. You can create a labeled LOOP statement in the following steps:

1. Write a valid LOOP, FOR LOOP, or WHILE LOOP statement.
2. Create a statement label by enclosing an SQL identifier (that is not already the name of a label in the same SPL routine) between angle brackets (`<<loop_label>>`) immediately before the first line of the LOOP, FOR LOOP, or WHILE LOOP statement.
3. Enter the same SQL identifier, but without angle bracket delimiters, immediately after the END LOOP keywords that terminate the statement, which is now a labeled loop statement.

One advantage of labeled LOOP statements is that they can be referenced in EXIT statements. When the EXIT *label* statement executes, program control passes from the EXIT statement to the statement that follows the specified loop label.

In the following example, a labeled WHILE LOOP loop, whose loop label identifier is **endo**, is part of the statement block of a labeled LOOP statement whose loop label identifier is **voort**. If the conditional EXIT statement `EXIT endo WHEN x = 7`: detects that its condition is true, program control passes to the `LET x = x + 1` statement that follows the `END LOOP endo`statement. If the conditional EXIT statement `EXIT voort WHEN x > 9`: detects that its condition is true, program control passes to the `LET x = x + 1` statement that follows the `END LOOP voort`statement, and the value of x is not incremented by the LET statement,

```
<<voort>>
LOOP
   LET x = x+1;
   <<endo>>
   WHILE ( i < 10 ) LOOP
      LET x = x+1;
         EXIT endo WHEN x = 7;
         EXIT voort WHEN x > 9;
         END LOOP endo;
      LET x = x+1;
   END LOOP voort;
```

uses FOR statement syntax to specify a variable and a range of values that the variable can take. The loop iterates until the specified limit to these values is reached, or until control is transferred outside the loop, as by the unconditional EXIT statement in the following example:

```
FOR i IN (1 TO 5) LOOP
   IF i = 5 THEN EXIT;
   ELSE
   CONTINUE;
END LOOP;
```

In the FOR LOOP statement, the FOR keyword can follow the EXIT or CONTINUE keyword, but the FOR keyword is not required, as it is in an ordinary FOR statement.

The following example replaces the IF statement with a functionally equivalent conditional EXIT statement:

```
FOR i IN (1 TO 5) LOOP
   EXIT WHEN i = 5;
```

```
    END LOOP;
```

Example

Related Statements

<< Label >> statement on page 913, FOR on page 931, WHILE on page 961

ON EXCEPTION

Use the ON EXCEPTION statement to specify actions to be taken for any error, or for a list of one or more specified errors, during execution of a statement block.

Syntax

(explicit id unique_716_Connect_42_onex002) unique_716_Connect_42_onex002 **ON EXCEPTION** [**IN**(*error_number*)] [**SET***SQL_error_var* [{ *, ISAM_error_var* | *, error_data_var* }]] <Statement Block> **END EXCEPTION** [**WITH RESUME**] [;]

Element	Description	Restrictions	Syntax
error_data_var	SPL variable to receive a string returned by an SQL error or by a user-defined exception	Must be a character type to receive the error information. Must be valid in current statement block.	Identifier on page 1231
error_number	SQL error number or a number defined by a RAISE EXCEPTION statement that is to be trapped	Must be of integer type. Must be valid in current statement block.	Literal Number on page 1198
ISAM_error_var	SPL variable that receives the ISAM error number of the exception raised	Same as for *error_number*	Identifier on page 1231
SQL_error_var	SPL variable that receives the SQL error number of the exception raised	Same as for *ISAM_error_var*	Identifier on page 1231

Usage

The ON EXCEPTION statement, together with the RAISE EXCEPTION statement, provides an error-trapping and error-recovery mechanism for SPL routines. ON EXCEPTION can specify the errors that you want to trap as the SPL routine executes, and specifies the action to take if the error occurs within the statement block. The ON EXCEPTION statement can list one or more specific error numbers in the IN clause, or it can trap all errors (or any error) if the IN clause is omitted.

A statement block can include more than one ON EXCEPTION statement. The exceptions that are trapped can be either system-defined or user-defined.

The scope of the ON EXCEPTION statement is the statement block that contains it, and any statement blocks that are nested within that statement block, unless one of the nested statement blocks provides an ON EXCEPTION statement that overrides the outer one.

When an exception is trapped, the error status is cleared.

If you specify a variable to receive an ISAM error, but no accompanying ISAM error exists, a zero (0) is assigned to the variable. If you specify a variable to receive the error text, but none exists, the variable stores an empty string.

No ON EXCEPTION Support in Triggered Actions

The ON EXCEPTION statement has no effect when it is issued from an SPL routine in the following calling contexts:

- in a trigger routine,
- in the Action clause or the Correlated Action clause of a trigger on a table,
- in the Action clause of an INSTEAD OF trigger on a view.

When a UDR includes ON EXCEPTION in any of these contexts, the database server ignores the ON EXCEPTION statement.

Placement of the ON EXCEPTION statement

The ON EXCEPTION statement is a declarative statement, not an executable statement. For this reason, ON EXCEPTION must follow immediately after any DEFINE statements, and must precede any executable statement within the same SPL statement block.

Because the body of the SPL routine is a statement block, the ON EXCEPTION statement often appears at the top of the routine, and applies to all of the code in the routine.

The following example positions an ON EXCEPTION statement so that a FOREACH statement can continue processing rows after an error occurs. Procedure X( ) reads customer numbers from table A and inserts them into table B. Because the INSERT statement is in scope of the ON EXCEPTION statement, any error during an INSERT operation causes control of execution to move to the next row of the FOREACH cursor, without terminating the FOREACH loop.

```
CREATE PROCEDURE X()

    DEFINE v_cust_num CHAR(20);

    FOREACH cs_insert FOR SELECT cust_num INTO v_cust_num FROM A
        BEGIN
            ON EXCEPTION
            END EXCEPTION WITH RESUME;
            INSERT INTO B(cust_num) VALUES(v_cust_num);
        END
    END FOREACH

END PROCEDURE
```

In the next example, function add_salesperson( ) inserts a set of values into a table. If the table does not exist, it is created, and the values are inserted. The function also returns the total number of rows in the table after the insert occurs:

```
CREATE FUNCTION add_salesperson(last CHAR(15), first CHAR(15))
   RETURNING INT;
   DEFINE x INT;
   ON EXCEPTION IN (-206) -- If no table was found, create one
      CREATE TABLE emp_list
```

```
          (lname CHAR(15),fname CHAR(15), tele CHAR(12));
      INSERT INTO emp_list VALUES -- and insert values
          (last, first, '800-555-1234');
   END EXCEPTION WITH RESUME;
   INSERT INTO emp_list VALUES (last, first, '800-555-1234');
   SELECT count(*) INTO x FROM emp_list;
   RETURN x;
END FUNCTION;
```

When an error occurs, the database server searches for the last ON EXCEPTION statement that traps the error code. If the database server finds no pertinent ON EXCEPTION statement, the error code is passed back to the calling context (the SPL routine, application, or interactive user), and execution terminates.

In the previous example, the minus sign (-) is required in the IN clause that specifies error -206; most error codes are negative integers.

The next example uses two ON EXCEPTION statements with the same error number so that error code 691 can be trapped in two levels of nesting. All of the DELETE statements except the one that is marked `{ 6 }` are within the scope of the first ON EXCEPTION statement. The DELETE statements that are marked `{ 1 }` and `{ 2 }` are within the scope of the inner ON EXCEPTION statement:

```
CREATE PROCEDURE delete_cust (cnum INT)
   ON EXCEPTION  IN (-691)    -- children exist
      BEGIN -- Begin-end so no other DELETEs get caught in here.
         ON EXCEPTION IN (-691)
            DELETE FROM another_child WHERE num = cnum;   { 1 }
            DELETE FROM orders WHERE customer_num = cnum; { 2 }
         END EXCEPTION -- for error -691
         DELETE FROM orders WHERE customer_num = cnum;    { 3 }
      END
      DELETE FROM cust_calls WHERE customer_num = cnum;   { 4 }
      DELETE FROM customer WHERE customer_num = cnum;     { 5 }
   END EXCEPTION
   DELETE FROM customer WHERE customer_num = cnum;        { 6 }
END PROCEDURE
```

Using the IN Clause to Trap Specific Exceptions

An error is trapped if the SQL error code or the ISAM error code matches an exception code in the list of error numbers. The search through the list of errors begins from the left and stops with the first match. You can use a combination of an ON EXCEPTION statement without an IN clause and one or more ON EXCEPTION statements with an IN clause. When an error occurs, the database server searches for the last declaration of the ON EXCEPTION statement that traps the particular error code.

```
CREATE PROCEDURE ex_test()
   DEFINE error_num INT;
   ...
   ON EXCEPTION SET error_num
   -- action C
   END EXCEPTION
   ON EXCEPTION IN (-300)
   -- action B
   END EXCEPTION
   ON EXCEPTION IN (-210, -211, -212) SET error_num
```

```
    -- action A
    END EXCEPTION
```

A summary of the sequence of statements in the previous example would be:

1. Test for an error.
2. If error `-210`, `-211`, or -`212` occurs, take action `A`.
3. If error `-300` occurs, take action `B`.
4. If any other error occurs, take action `C`.

Receiving Error Information in the SET Clause

If you use the SET clause, when an exception occurs, the SQL error code and (optionally) the ISAM error code are inserted into the variables that are specified in the SET clause. If you provide an *error_data_var*, any error text that the database server returns is put into the *error_data_var*. Error text includes information such as the offending table or column name.

Forcing Continuation of the Routine

The first example in Placement of the ON EXCEPTION statement on page 950 includes the WITH RESUME keyword to specify that if the ON EXCEPTION statement traps an error, execution of the FOREACH loop resumes on the next row of the **cs_insert** cursor, the row immediately following the row on which the error was raised. If an error is issued on the last row of the active set, the procedure exits. After procedure **X** completes execution, table **B** contains a copy of every customer number in table **A** on which no error was issued during the INSERT operation.

The second example in Placement of the ON EXCEPTION statement on page 950 uses the WITH RESUME keyword to indicate that after the statement block in the ON EXCEPTION statement executes, execution is to continue at the `SELECT COUNT(*) FROM emp_list` statement, which is the line following the line that raised the error. For this function, the result is that the count of salespeople names occurs even if the error occurred.

Continuing Execution After an Exception Occurs

If you omit the WITH RESUME keywords, the next statement that executes after an exception occurs depends on the placement of the ON EXCEPTION statement, as the following scenarios describe:

- If the ON EXCEPTION statement is inside a statement block with a BEGIN and an END keyword, execution resumes with the first statement (if any) after that BEGIN ... END block. That is, it resumes after the scope of the ON EXCEPTION statement.
- If the ON EXCEPTION statement is inside a loop (FOR, WHILE, FOREACH), the rest of the loop is skipped, and execution resumes with the next iteration of the loop.
- If no statement or block, but only the SPL routine, contains the ON EXCEPTION statement, the routine executes a RETURN statement with no arguments, returning a successful status and no values.

To prevent an infinite loop, if an error occurs during execution of the statement block, then the search for another ON EXCEPTION statement to trap the error does not include the current ON EXCEPTION statement.

Related Statements

RAISE EXCEPTION on page 953

RAISE EXCEPTION

Use the RAISE EXCEPTION statement to simulate the generation of an error.

Syntax

```
RAISE EXCEPTION SQL_error_var [ , ISAM_error [ , error_text ] ] ;
```

Element	Description	Restrictions	Syntax
error_text	SPL variable or expression that contains error message text for error -746	Must be a character data type and be valid in the statement block	Identifier on page 1231; Expression on page 1003
ISAM_error	SPL variable or other expression that represents an ISAM error number. The default is `0`.	Must return a value in SMALLINT range. You can specify a unary minus sign before error number.	Expression on page 1003
SQL_error	SPL variable or other expression that represents an SQL error number	Same as for *ISAM_error*	Expression on page 1003

Usage

Use the RAISE EXCEPTION statement to simulate an error or to generate an error with a custom message. An ON EXCEPTION statement can trap the generated error.

If you omit *ISAM_error*, the database server sets the ISAM error code to zero (`0`) when the exception is raised. If you want to specify *error_text* but not specify a value for *ISAM_error*, specify zero (`0`) as the value of *ISAM_error*.

The RAISE EXCEPTION statement can raise either system-generated exceptions or user-generated exceptions. For example, the following statement raises the error number `-208`:

```
RAISE EXCEPTION -208, 0;
```

Here the minus (-) symbol is required after the EXCEPTION keyword for error -208; most error codes are negative integers.

Special Error Number -746

The special error number `-746` allows you to produce a customized message. For example, the following statement raises the error number `-746` and returns the quoted text:

```
RAISE EXCEPTION -746, 0, 'You broke the rules';
```

In the following example, a negative value for **alpha** raises exception `-746` and provides a specific message that describes the problem. The code should contain an ON EXCEPTION statement that traps for an exception of `-746`.

```
FOREACH SELECT c1 INTO alpha FROM sometable
IF alpha < 0 THEN
RAISE EXCEPTION -746, 0, 'a < 0 found' -- emergency exit
END IF
END FOREACH
```

When the SPL routine executes and the IF condition is met, the database server returns the following error:

```
-746: a < 0 found.
```

For more information about the scope and compatibility of exceptions, see ON EXCEPTION on page 949.

Related Statements

ON EXCEPTION on page 949

RETURN

Use the RETURN statement to specify what values (if any) the SPL function returns to the calling context.

Syntax



Usage

In Informix®, for backward compatibility, you can use the RETURN statement inside a CREATE PROCEDURE statement to create an SPL function. By only using RETURN in CREATE FUNCTION statements, however, you can maintain the convention of using CREATE FUNCTION to define routines that return a value, and CREATE PROCEDURE for other routines.

All RETURN statements in the SPL function must be consistent with the RETURNING clause of the CREATE FUNCTION (or CREATE PROCEDURE) statement that defines the function. Any RETURN list of expressions must match in cardinality (and be of data types compatible with) the ordered list of data types in the RETURNING clause of the function definition.

Alternatively, however, the RETURN statement can specify no expressions, even if the RETURNING clause lists one or more data types. In this case, a RETURN statement that specifies no expression is equivalent to returning the expected number of NULL values to the calling context. A RETURN statement without any expressions exits only if the SPL function is declared as not returning any values. Otherwise it returns NULL values.

The following SPL function has two valid RETURN statements:

```
CREATE FUNCTION two_returns (stockno INT)  RETURNING CHAR (15);
   DEFINE des CHAR(15);
   ON EXCEPTION (-272)     -- if user does not have select privilege
      RETURN;              -- return no values.
   END EXCEPTION;
   SELECT DISTINCT descript INTO des FROM stock
      WHERE stock_num = stockno;
   RETURN des;
END FUNCTION;
```

A program that calls the function in the previous example should test whether no values are returned and act accordingly.

WITH RESUME Keyword

If you use the WITH RESUME keywords, then after the RETURN statement completes execution, the next invocation of the SPL function (upon the next FETCH or FOREACH statement) starts from the statement that follows the RETURN statement. Any function that executes a RETURN WITH RESUME statement must be invoked within a FOREACH loop, or else in the FROM clause of a SELECT statement. If an SPL routine executes a RETURN WITH RESUME statement, a FETCH statement in the application can call the SPL routine.

The following example shows a cursor function that another UDR can call. After the RETURN WITH RESUME statement returns each value to the calling UDR or program, the next line of **series** executes the next time **series** is called. If the variable **backwards** equals zero (0), no value is returned to the calling UDR or program, and execution of **series** stops:

```
CREATE FUNCTION series (limit INT, backwards INT) RETURNING INT;
   DEFINE i INT;
   FOR i IN (1 TO limit)
      RETURN i WITH RESUME;
   END FOR;
   IF backwards = 0 THEN
      RETURN;
   END IF;
   FOR i IN (limit TO 1 STEP -1)
      RETURN i WITH RESUME;
   END FOR;
END FUNCTION; -- series
```

Returning Values from Another Database

If an SPL function uses the Return clause to return values from another database of the local Informix® instance, the following data types are supported as the returned data type:

- Built-in data types that are not opaque
- Most of the *built-in opaque data types*, as listed in Data Types in Cross-Database Transactions on page 700
- DISTINCT of the built-in types that are referenced in the two lines above
- DISTINCT of any DISTINCT data type in this list
- Any opaque user-defined type (UDT) that is cast explicitly to one of the built-in data types in this list.

The definitions of the UDF and of the type hierarchies, casts, DISTINCT types, and UDTs must be exactly the same in each of the participating databases. The same data-type restrictions apply to a value that an external function returns from another database of the local Informix® instance. For more information about data types that are supported in distributed operations across two or more databases of the same database server, see Data Types in Cross-Database Transactions on page 700.

UDRs can return only the following data types from tables in databases of other database servers:

- Any non-opaque built-in data type
- BOOLEAN
- LVARCHAR
- DISTINCT of non-opaque built-in types
- DISTINCT of BOOLEAN

- DISTINCT of LVARCHAR
- DISTINCT of any DISTINCT type that appears in this list.

UDRs can return these DISTINCT types from databases of other Informix® instances only if the DISTINCT types are cast explicitly to built-in types. The definitions of the DISTINCT data types, their type hierarchies, and their casts must be exactly the same in all databases that participate in the distributed operations. For queries or other DML operations in cross-server UDRs that use the data types in the preceding list as parameters or as returned data types, the UDR must be defined in each participating database, and the participating Informix® instances must support the data type as a returned value in cross-server operations.

For additional information about the data types that Informix® can access in distributed operations, see Data Types in Distributed Queries on page 700.

External Functions and Iterator Functions

In an SPL program, you can use a C or Java™ language external function as an expression in a RETURN statement, provided that the external function is not an iterator function. An *iterator function* is an external function that returns one or more rows of data (and therefore requires a cursor to execute).

SPL iterator functions must include the RETURN WITH RESUME statement. For information about using an iterator function with a virtual table interface in the FROM clause of a query, see Iterator Functions on page 720.

SYSTEM

Use the SYSTEM statement to issue an operating-system command from within an SPL routine.

Syntax

```
SYSTEM { expression | SPL_var } ;
```

Element	Description	Restrictions	Syntax
expression	Evaluates to a user-executable operating-system command	You cannot specify that the command run in the background	Operating-system dependent
SPL_var	SPL variable containing a command	Must be of a character data type	Identifier on page 1231;

Usage

If the specified *expression* is not a character expression, it is converted to a character expression and passed to the operating system for execution.

The command that SYSTEM specifies cannot run in the background. The database server waits for the operating system to complete execution of the command before it continues to the next statement in the SPL routine. The SPL routine cannot use any returned values from the command.

If the operating-system command fails (that is, returns a nonzero status for the command), an exception is raised that contains the returned operating-system status as the error code and an appropriate SQL error code.

A rollback does not terminate a system call, so a suspended transaction can wait indefinitely for the call to return. For instructions on recovery from a deadlock during a long transaction rollback, see the *HCL® Informix® Administrator's Guide*.

The dynamic log feature of Informix® automatically adds log files until the long transaction completes or rolls back successfully.

In DBA- and owner-privileged SPL routines that contain SYSTEM statements, the command runs with the access privileges of the user who executes the routine.

Executing the SYSTEM statement on UNIX™

In SPL procedures for UNIX™ platforms, a specification that evaluates to a valid UNIX™ operating system command must immediately follow the SYSTEM keyword.

Both of the program fragments that follow use the SYSTEM statement of SPL to send a message to the system administrator.

- In the first example, the **sensitive_update** routine defines an SPL variable called **mailcall** to store a character string that specifies the name of the **mail** utility, and the user ID of the message recipient, and the message text.
- In the second example, the **sensitive_update2** routine similarly invokes the **mail** utility with a SYSTEM statement. The expression constructs a valid command line by concatenating three quoted strings and the SPL variables **user1** and **user2** to send to the system administrator a file called **violations_file**.

Sending email using the SYSTEM statement

The SYSTEM statement in the following example of an SPL routine causes the UNIX™ operating system to send a mail message to the system administrator whose user ID is **headhoncho**:

```
CREATE PROCEDURE sensitive_update()
   ...
   LET mailcall = 'mail headhoncho < alert';
   -- code to execute if user tries to execute a specified
   -- command, then sends email to system administrator
   SYSTEM mailcall;
   ...
END PROCEDURE; -- sensitive_update
```

You can use a double-pipe symbol (`||`) to concatenate expressions within a SYSTEM statement, as the following example shows:

```
CREATE PROCEDURE sensitive_update2()
   DEFINE user1 char(15);
   DEFINE user2 char(15);
   LET user1 = 'joe';
   LET user2 = 'mary';
   ...
   -- code to execute if user tries to execute a specified
   -- command, then sends email to system administrator
```

```
   SYSTEM 'mail -s violation' || user1 || ' ' || user2
                || '< violation_file';
   ...
END PROCEDURE; --sensitive_update2
```

In both examples above, blank spaces separate elements of the command line, so the expression that follows the SYSTEM keyword evaluates to a character string that conforms to the syntax requirements of the operating system mail utility.

Executing the SYSTEM statement on Windows™

On Windows™ systems, any SYSTEM statements in an SPL routine are executed only if the current user who is executing the SPL routine has logged on with a password.

The database server must have the password and login name of the user in order to execute a command on behalf of that user.

The first SYSTEM statement in the following example of an SPL routine causes Windows™ to send an error message to a temporary file and to put the message in a system log that is sorted alphabetically. The second SYSTEM statement causes the operating system to delete the temporary file:

```
CREATE PROCEDURE test_proc()
   ...
   SYSTEM 'type errormess101 > %tmp%tmpfile.txt |
         sort >> %SystemRoot%systemlog.txt';
   SYSTEM 'del %tmp%tmpfile.txt';
   ...
END PROCEDURE; --test_proc
```

The expressions that follow the SYSTEM statements in this example contain variables **%tmp%** and **%SystemRoot%** that are defined by Windows™.

Setting Environment Variables in SYSTEM Commands

About this task

When the operating-system command that SYSTEM specifies is executed, no guarantee exists that any environment variables that the user application sets are passed to the operating system. If you set an environment variable in a SYSTEM command, the setting is only valid during that SYSTEM command.

To avoid this potential problem, the following method is recommended to ensure that any environment variables that the user application requires are carried forward to the operating system.

To Change Environment Settings for an Operating System Command

1. Create a shell script (on UNIX™ systems) or a batch file (on Windows™ platforms) that sets up the desired environment and then executes the operating system command.
2. Use the SYSTEM command to execute the shell script or batch file.

Results

This solution has an additional advantage: if you subsequently need to change the environment, you can modify the shell script or the batch file without needing to recompile the SPL routine.

For information about operating system commands that set environment variables, see the *HCL® Informix® Guide to SQL: Reference*.

TRACE

Use the TRACE statement to control the generation of debugging output.

Syntax



Usage

The TRACE statement generates output that is sent to the file that the SET DEBUG FILE TO statement specifies. Tracing writes to the debug file the current values of the following program objects:

- SPL variables
- Routine arguments
- Return values
- SQL error codes
- error codes

The output of each executed TRACE statement appears on a separate line.

If you use the TRACE statement before you specify a DEBUG file to contain the output, an error is generated.

Any routine that the SPL routine calls inherits the trace state. That is, a called routine (on the same database server) assumes the same trace state (ON, OFF, or PROCEDURE) as the calling routine. The called routine can set its own trace state, but that state is not passed back to the calling routine.

A routine that is executed on a remote database server does not inherit the trace state.

TRACE ON

If you specify the keyword ON, all statements are traced. The values of variables (in expressions or otherwise) are printed before they are used. To turn tracing ON implies tracing both routine calls and statements in the body of the routine.

TRACE OFF

If you specify the keyword OFF, all tracing is turned off.

TRACE PROCEDURE

If you specify the keyword PROCEDURE, only the routine calls and return values, but not the body of the routine, are traced.

The TRACE statement supports no ROUTINE or FUNCTION keywords. Use the TRACE PROCEDURE keywords when the SPL routine that you trace is a function.

Displaying Expressions

You can use the TRACE statement with a quoted string or an expression to display values or comments in the output file. If the expression is not a literal expression, the expression is evaluated before it is written to the output file.

You can use the TRACE statement with an expression even if you used a TRACE OFF statement earlier in a routine. You must first, however, use the SET DEBUG statement to establish a trace output file.

The next example uses a TRACE statement with an expression after using a TRACE OFF statement. The example uses UNIX™ file naming conventions:

```
CREATE PROCEDURE tracing ()
   DEFINE i INT;
BEGIN
   ON EXCEPTION IN (1)
   END EXCEPTION; -- do nothing
   SET DEBUG FILE TO '/tmp/foo.trace';
   TRACE OFF;
   TRACE 'Forloop starts';
   FOR i IN (1 TO 1000)
      BEGIN
         TRACE 'FOREACH starts';
         FOREACH SELECT...INTO a FROM t
            IF <some condition> THEN
               RAISE EXCEPTION 1    -- emergency exit
            END IF
         END FOREACH                -- return some value
      END
   END FOR                          -- do something
END;
END PROCEDURE
```

Example Showing Different Forms of TRACE

The following example shows several different forms of the TRACE statement. The example uses Windows™ file naming conventions:

```
CREATE PROCEDURE testproc()
   DEFINE i INT;
   SET DEBUG FILE TO 'C:\tmp\test.trace';
   TRACE OFF;
   TRACE 'Entering foo';
   TRACE PROCEDURE;
   LET i = test2();

   TRACE ON;
   LET i = i + 1;

   TRACE OFF;
   TRACE 'i+1 = ' || i+1;
   TRACE 'Exiting testproc';
```

```
   SET DEBUG FILE TO 'C:\tmp\test2.trace';

END PROCEDURE
```

Looking at the Traced Output

About this task

To see the traced output, use a text editor or similar utility to display or read the contents of the file.

WHILE

Use the WHILE statement to establish a loop with variable end conditions.

Syntax

(explicit id unique_1312_Connect_42_whil001) unique_1312_Connect_42_whil001(explicit id unique_1312_Connect_42_whil002) unique_1312_Connect_42_whil002(explicit id unique_1312_Connect_42_whil003) unique_1312_Connect_42_whil003 [<<*label* >>] **WHILE** <Condition>[] { <Statement Block>[] **END WHILE** [*label*[]] | **LOOP** <Statement Block>[] **END LOOP** [*label*[]] } [;]

Element	Description	Restrictions	Syntax
label	Name of the loop label for this loop	Must be unique among labels in this SPL routine	Identifier on page 1231

Usage

The *condition* is evaluated before the *statement block* first runs and before each subsequent iteration. Iterations® continue as long as the *condition* remains true. The loop terminates when the *condition* evaluates to not true.

If any expression within the *condition* evaluates to NULL, the *condition* becomes not true unless you are explicitly testing for NULL with the IS NULL operator.

If an expression within the *condition* has an UNKNOWN value because it references uninitialized SPL variables, an immediate error results. In this case, the loop terminates, raising an exception.

Example of WHILE Loops in an SPL Routine

The following example illustrates the use of WHILE loops in an SPL routine. In the SPL procedure, **simp_while**, the first WHILE loop executes a DELETE statement. The second WHILE loop executes an INSERT statement and increments the value of an SPL variable.

```
CREATE PROCEDURE simp_while()
   DEFINE i INT;
   WHILE EXISTS (SELECT fname FROM customer
       WHERE customer_num > 400)
      DELETE FROM customer WHERE id_2 = 2;
   END WHILE;
```

```
   LET i = 1;
   WHILE i < 10
      INSERT INTO tab_2 VALUES (i);
      LET i = i + 1;
   END WHILE;
END PROCEDURE;
```

Labeled WHILE Loops

To create a labeled WHILE loop, you can declare a loop label before the initial WHILE keyword, and repeat the label after the END WHILE keywords, as in the two WHILE loops of the following example:

```
CREATE PROCEDURE ex_cont_ex()
   DEFINE i,s,j, INT;
   <<while_jlab>>
   WHILE j < 20
      IF j > 10 THEN
         CONTINUE WHILE;
      END IF
      LET i,s = j,0;
      <<while_slab>>
      WHILE i > 0
         LET i = i -1;
         IF i = 5 THEN
            EXIT while_jlab;
         END IF
      END WHILE while_slab
   END WHILE while_jlab
END PROCEDURE;
```

Here the `EXIT while_jlab` statement has the same effect that the `EXIT` or `EXIT FOR` keywords would have, terminating both the outer WHILE loop and the routine. In this example, the statement that includes the `EXIT while_jlab` statement has the same effect that `EXIT while_jlab WHEN i = 5` would have.

You can also label a LOOP statement that begins with a loop <<*label*>> specification that immediately precedes the initial WHILE keyword and *condition*. In this type of loop, the CONTINUE LOOP, EXIT LOOP, and END LOOP keywords replace the CONTINUE WHILE, EXIT WHILE, and END WHILE keywords. Both the LOOP and WHILE keywords are optional after the CONTINUE and EXIT keywords, but the END LOOP keywords are required in SPL loop statements that include the LOOP keyword.

You can use similar syntax to create an unlabeled loop that omits the <<*label*>> declaration that immediately precedes the WHILE *condition* specification. In this case, you must also omit the undelimited loop label identifier that follows the END LOOP keywords. See the LOOP statement for a description and examples of these forms of labeled and unlabeled loop statements that enable you to combine WHILE statement syntax, with its condition-based number of loop iterations, with the "loop forever" syntax of the LOOP statement.

Related Statements

[<< Label >> statement on page 913](), [CONTINUE on page 919](), [EXIT on page 929](), [LOOP on page 945]()

Data types and expressions

These topics describe the data types and expressions that Informix®supports.

These fundamental syntax segments can appear in data definition language (DDL) and data manipulation language (DML) statements, and in other types of SQL statements. Some SPL statements can also specify data types or expressions. You can use these features of a relational or object-relational database in various contexts, such as to define the schema of a table, to specify the signature and arguments of a routine, or to represent or calculate specific data values.

Scope of Segment Descriptions

The description of each segment includes the following information:

- A brief introduction that explains the effect of the segment
- A syntax diagram that shows how to enter the segment correctly
- A table that explains the terms in the syntax diagram for which you must substitute names, values, or other specific information
- Rules of usage, typically including examples that illustrate these rules

If a segment consists of multiple parts, the segment description provides similar information about each part. Some descriptions conclude with references to related information in this document and in other documents.

Use of Segment Descriptions

The syntax diagram within each segment description is not a stand-alone diagram. Rather, it is a subdiagram of the syntax of the SQL statements (in SQL statements on page 18) or of SPL statements (in SPL statements on page 911) that can include the segment.

SQL or SPL syntax descriptions can refer to segment descriptions in two ways:

- A *subdiagram reference* in a syntax diagram can list a segment name and the page in this document where the segment description begins.
- The **Syntax** column of the table that immediately follows a syntax diagram can list a segment name and the page where the segment description begins.

If the syntax diagram for a statement includes a reference to a segment, turn to that segment description to see the complete syntax for the segment.

For example, if you want to write a CREATE VIEW statement that includes a *database* and *database server* qualifiers of the *view* name, first look up the syntax diagram for the CREATE VIEW statement on page 421. The table beneath that diagram refers to the Database Object Name segment for the syntax of *view.* Then use the Database Object Name segment syntax to enter a valid CREATE VIEW statement that also specifies the *database* and *database server* name for the view. In the following example, the CREATE VIEW statement defines a view called **name_only** in the **sales** database on the **boston** database server:

```
CREATE VIEW sales@boston:name_only AS
      SELECT customer_num, fname, lname FROM customer;
```

Besides the Data Types and Expressions syntax segments that this chapter documents, Other syntax segments on page 1209 provides additional syntax segments that are referenced in the syntax diagrams of this document.

Data type and expression segments

Data type and expression segments can appear in SQL statements.

Data type and expression segments can include the following items:

- Data Type
- DATETIME Field Qualifier
- INTERVAL Field Qualifier
- Expression
- Aggregate Expression
- AVG, COUNT, MAX, MIN, SUM, RANGE, STDEV, VARIANCE, *and* User-Defined Aggregates
- Arithmetic Expressions
- Binary (+, -, *, /) Operators, Operator Functions, *and* Unary (+, -) Operators
- Cast Expressions
- CAST function *and* Cast (::) Operator
- Collection Subquery
- Column Expressions
- Column Name, ROWID, *and* Substring ([...]) Operator
- CONCAT Function *and* Concatenation (||) Operator
- Condition Segment *and* Conditional Expressions
- Comparison Condition: AND, OR, NOT, BETWEEN, IS NULL, LIKE, MATCHES, *and* Relational Operators
- Condition with Subquery: IN, EXISTS, ALL, ANY, *and* SOME Operators
- Boolean UDF
- CASE Expressions
- NVL Function
- DECODE Function
- Constant Expressions: CURRENT, SYSDATE, TODAY, DBSERVERNAME, SITENAME, UNITS, CURRENT_USER, *and* USER
- Literal Value
- Literal Collection
- Literal DATETIME
- Literal INTERVAL
- Literal Number
- Literal Row
- Quoted String
- Constructor Expressions
- Collection Constructor
- ROW Constructor Function Expressions
- Algebraic Functions: ABS, MOD, POW, power ROOT, ROUND, SQRT, *and* TRUNC Functions
- CARDINALITY Function

- DBINFO Function
- Encryption and Decryption Functions: DECRYPT_BINARY, DECRYPT_CHAR, ENCRYPT_AES, ENCRYPT_TDES, *and* GETHINT Functions
- Exponential and Logarithmic Functions: EXP, LOGN, *and* LOG10 Functions
- HEX Function
- Hierarchical Query Operators and Functions: CONNECT_BY_ROOT, PRIOR, *and* SQL_CONNECT_BY_PATH
- IFX_ALLOW_NEWLINE Function
- Length Functions: CHARACTER_LENGTH, CHAR_LENGTH, LENGTH, *and* OCTET_LENGTH Functions
- Sequence Operators: CURRVAL, NEXTVAL
- Smart Large Object Functions: FILETOBLOB, FILETOCLOB, LOCOPY, *and* LOTOFILE Functions
- String-Manipulation Functions: LPAD, RPAD, TRIM, REPLACE, SUBSTR, SUBSTRING, INITCAP, LOWER, *and* UPPER Functions
- Time Functions: DATE, DAY, EXTEND, MDY, MONTH, TO_CHAR, TO_DATE, WEEKDAY, *and* YEAR Functions
- Trigger-Type Boolean Operators: DELETING, INSERTING, SELECTING, *and* UPDATING
- Trigonometric Functions: ACOS, ASIN, ATAN, ATAN2, COS, SIN, *and* TAN Functions
- User-Defined Functions
- Statement-Local Variable Expressions

You can also use host variables or SPL variables as expressions. For an alphabetic list of expressions with page references, see List of Expressions on page 1005.

Collection Subquery

You can use a Collection Subquery to create a MULTISET collection from the results of a subquery. This syntax is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_1010_Connect_42_colsub001) unique_1010_Connect_42_colsub001

Collection Subquery

" [] **MULTISET** "

" ({ *subquery* | **SELECT ITEM** *singleton_select* }) "

Element	Description	Restrictions	Syntax
singleton_select	Subquery returning exactly one row	Subquery cannot repeat the SELECT keyword, nor include the ORDER BY clause	SELECT statement on page 690
subquery	Embedded query	Cannot contain the ORDER BY clause	SELECT statement on page 690

Usage

The MULTISET and SELECT ITEM keywords have the following significance:

- MULTISET specifies a collection of elements that can contain duplicate values, but that has no specific order of elements.
- SELECT ITEM supports only one expression in the projection list. You cannot repeat the SELECT keyword in the singleton subquery.

You can use a collection subquery in the following contexts:

- The Projection clause and WHERE clause of the SELECT statement
- The VALUES clause of the INSERT statement
- The SET clause of the UPDATE statement
- Wherever you can use a collection expression (that is, any expression that evaluates to a single collection)
- As an argument passed to a user-defined routine

The following restrictions apply to a collection subquery:

- The Projection clause cannot contain duplicate column (field) names.
- It cannot contain aliases for table names. (But it can use aliases for column (field) names, as in some of the examples that follow.)
- It is read-only.
- It cannot be opened twice.
- It cannot contain NULL values.
- It cannot contain syntax that attempts to seek within the subquery.

A collection subquery returns a multiset of unnamed ROW data types. The fields of this ROW type are elements in the projection list of the subquery. Examples that follow access the tables and the ROW types that these statements define:

```
CREATE ROW TYPE rt1 (a INT);
CREATE ROW TYPE rt2 (x int, y rt1);
CREATE TABLE tab1 (col1 rt1, col2 rt2);
CREATE TABLE tab2 OF TYPE rt1;
CREATE TABLE tab3 (a ROW(x INT));
```

The following examples of collection subqueries return the MULTISET collections that are listed to the right of the subquery.

Collection Subquery	Resulting Collections
MULTISET (SELECT * FROM tab1)...	MULTISET(ROW(col1 rt1, col2 rt2))
MULTISET (SELECT col2.y FROM tab1)...	MULTISET(ROW(y rt1))
MULTISET (SELECT * FROM tab2)...	MULTISET(ROW(a int))
MULTISET(SELECT p FROM tab2 p)...	MULTISET(ROW(p rt1))
MULTISET (SELECT * FROM tab3)...	MULTISET(ROW(a ROW(x int)))

The following is another collection subquery:

```
SELECT f(MULTISET(SELECT * FROM tab1 WHERE tab1.x = t.y))
   FROM t WHERE t.name = 'john doe';
```

The following collection subquery includes the UNION operator:

```
SELECT f(MULTISET(SELECT id FROM tab1
UNION
SELECT id FROM tab2 WHERE tab2.id2 = tab3.id3)) FROM tab3;
```

Table expressions in the FROM clause

Informix® supports ANSI/ISO standard syntax for table expressions in the FROM clause of SELECT queries and subqueries as a substitute for the Informix®-extension collection subquery syntax. The keywords TABLE and MULTISET are required in version 10.00 and in earlier releases. These extensions to the ANSI/ISO standard for SQL are supported but are no longer required for collection subqueries in the FROM clause of SELECT statements.

The following two queries return the same result set, but only the second query complies with the ANSI/ISO standard:

```
SELECT * FROM TABLE(MULTISET(SELECT col1 FROM tab1 WHERE col1 = 100))
   AS vtab(c1),
   (SELECT col1 FROM tab1 WHERE col1 = 10) AS vtab1(vc1) ORDER BY c1;

SELECT * FROM (SELECT col1 FROM tab1 WHERE col1 = 100) AS vtab(c1),
   (SELECT col1 FROM tab1 WHERE col1 = 10) AS vtab1(vc1)
   ORDER BY c1;
```

The same SELECT statement can combine instances of both the Informix®-extension and ANSI/ISO syntax for collection subqueries:

```
SELECT * FROM (select col1 FROM tab1 WHERE col1 = 100) AS vtab(c1),
   TABLE(MULTISET(SELECT col1 FROM tab1 WHERE col1 = 10)) AS vtab1(vc1)
   ORDER BY c1;
```

The collection subquery must be delimited by parentheses in both formats, but the outer set of parentheses (( )) that immediately follows the TABLE keyword and encloses the MULTISET collection subquery specification is an extension to the ANSI/ISO syntax. This ANSI/ISO syntax is valid only in the FROM clause of the SELECT statement. You cannot omit these keywords and parentheses from a collection subquery specification in any other context.

Important:

Collection subqueries in the FROM clause cannot include correlated table references, and cannot include the LATERAL keyword.

Condition

Use a condition to test whether data meets certain qualifications. Use this segment wherever you see a reference to a condition in a syntax diagram.

Syntax

(explicit id unique_62_Connect_42_cond004) unique_62_Connect_42_cond004(explicit id unique_62_Connect_42_cond005)

unique_62_Connect_42_cond005(explicit id unique_62_Connect_42_cond006) unique_62_Connect_42_cond006

Condition

" [NOT] { <Comparison Conditions> [] | <Condition with Subquery> [] | <User-Defined Function> [] } "

Element	Description	Restrictions	Syntax
Logical _Operator	Combines two conditions	Valid options are OR (= *logical union*) *or* AND (= *logical intersection*)	Conditions with AND or OR on page 984

Usage

A *condition* is a search criterion, optionally connected by the logical operators AND or OR. Conditions can be classified into the following categories:

- Comparison conditions (also called filters or Boolean expressions)
- Conditions with a subquery
- User-defined functions (Informix® only)

A condition can contain an aggregate function only if it is used in the HAVING clause of a SELECT statement or in the HAVING clause of a subquery.

No aggregate function can appear in a condition in the WHERE clause of a DELETE, SELECT, or UPDATE statement unless both of the following are `TRUE`:

- Aggregate is on a correlated column originating from a parent query.
- The WHERE clause appears in a subquery within a HAVING clause.

In , user-defined functions are not valid as conditions in the following contexts:

- In the HAVING clause of a SELECT statement
- In the definition of a check constraint

SPL routines are not valid as conditions in the following contexts:

- In the definition of a check constraint
- In the ON clause of a SELECT statement
- In the WHERE clause of a DELETE, SELECT, or UPDATE statement

External routines are not valid as conditions in the following contexts:

- In the definition of a check constraint
- In the ON clause of a SELECT statement
- In the WHERE clause of a DELETE, SELECT, or UPDATE statement
- In the WHEN clause of CREATE TRIGGER
- In the IF, CASE, or WHILE statements of SPL

Column Name

The Column Name segment can be an element in comparison conditions. The name of a column (or of one or more fields within a column of a ROW data type) is not the subject of the comparison, but the database server uses this SQL identifier to access the data value in the specified column or field of a row in a database table or view.

(explicit id unique_1396_Connect_42_coln003) unique_1396_Connect_42_coln003

Column Name

" [{ *table* . | *view* . | *synonym* . | *alias* . }] "

" { *column* | *row_column* [[] .*field*] } "

Element	Description	Restrictions	Syntax
alias	Temporary alternative name for table or view	Must be defined in the FROM clause of the SELECT statement	Identifier on page 1231
column	Name of a column	Must exist in the specified table	Identifier on page 1231
field	A field to compare in a ROW type column	Must be a component of *row-column name* or *field name* (for nested rows)	Identifier on page 1231
row_column	A column of type ROW	Must be an existing named ROW type or unnamed ROW type	Identifier on page 1231
synonym, *table*, *view*	Name of a synonym, table, or view	The *synonym* and the table or view to which it points must exist in the database	Identifier on page 1231

For more information on the meaning of the *column* name in these conditions, see the IS NULL and IS NOT NULL Conditions on page 974 and the LIKE and MATCHES Condition on page 975.

Quotation Marks in Conditions

When you compare a column expression with a constant expression in any comparison condition, observe the following rules:

- If the column has a numeric data type, do not enclose the constant expression between quotation marks.
- If the column has a character data type, enclose the constant expression between quotation marks.
- If the column has a time data type, enclose the constant expression between quotation marks.

Otherwise, you might get unexpected results.

The following example shows the correct use of quotation marks in comparison conditions. Here the **ship_instruct** column has a character data type, the **order_date** column has a date data type, and the **ship_weight** column has a numeric data type.

```
SELECT * FROM orders
   WHERE ship_instruct = 'express'
   AND order_date > '05/01/98'
   AND ship_weight < 30;
```

Relational-Operator Condition

A relational operator compares two expressions quantitatively.

For a list of the supported relational operators and their descriptions, see Relational Operator on page 1206.

The following examples show some relational-operator conditions:

```
city[1,3] = 'San'

o.order_date > '6/12/98'

WEEKDAY(paid_date) = WEEKDAY(CURRENT- (31 UNITS DAY))

YEAR(ship_date) < YEAR (TODAY)

quantity <= 3

customer_num <> 105

customer_num != 105
```

Operands in relational operator conditions cannot have UNKNOWN or NULL values. If an expression within the *condition* has an UNKNOWN value because it references an uninitialized variable, the database server raises an exception.

Conditions testing for NULL values

If any expression within the *condition* evaluates to NULL, the *condition* cannot be true, unless you are explicitly testing for NULL by using the IS NULL operator. For example, if the **paid_date** column has a NULL value, then neither of the following queries can retrieve that row:

```
SELECT customer_num, order_date FROM orders
   WHERE paid_date = '';

SELECT customer_num, order_date FROM orders
   WHERE NOT (paid_date !='');
```

You must use the IS NULL operator to test for a NULL value, as the next example shows.

```
SELECT customer_num, order_date FROM orders
   WHERE paid_date IS NULL;
```

The IS NULL operator and its logical inverse, the IS NOT NULL operator, are described in IS NULL and IS NOT NULL Conditions on page 974.

BETWEEN Condition

Use the BETWEEN condition to test whether the value of a numeric, character, or time expression is within a specified range.

(explicit id unique_1051_Connect_42_betw002) unique_1051_Connect_42_betw002

BETWEEN Condition

" <Expression> [NOT] BETWEEN <Expression> AND <Expression> "

Usage

NULL values cannot satisfy the condition. Neither of the expressions that define the range can evaluate to NULL.

The three expressions in a BETWEEN condition must satisfy these restrictions:

- All three expressions must evaluate to mutually compatible numeric, time, or character data types.
- The value of the expression that immediately follows the `BETWEEN` keyword must be less than the value of the expression that follows the `AND` keyword.

Numeric and time expressions in BETWEEN conditions

For number expressions, *less than* means to the left on the real line.

For DATE and DATETIME expressions, *less than* means earlier in time.

For INTERVAL expressions, *less than* means a shorter span of time.

Character expressions in BETWEEN conditions

For CHAR, VARCHAR, and LVARCHAR expressions, *less than* means *before* in code-set order.

For NCHAR and NVARCHAR expressions, *less than* means *before* in the localized collation order, if one exists; otherwise, *less than* means *before* in code-set order.

Locale-based collation order, if one is defined for the locale, is used for NCHAR and NVARCHAR expressions. So for NCHAR and NVARCHAR expressions, *less than* means *before* in the locale-based collation order. For more information on locale-based collation order and the NCHAR and NVARCHAR data types, see the *HCL® Informix® GLS User's Guide*.

For information on how relational operator expressions with NCHAR and NVARCHAR operands in databases that have the NLCASE INSENSITIVE property differ from their behavior in databases that are case sensitive, see the topic NCHAR and NVARCHAR expressions in case-insensitive databases.

The NOT keyword in BETWEEN conditions

For a BETWEEN condition to be `TRUE` depends on whether you include the `NOT` keyword.

- If you omit the `NOT` keyword, the BETWEEN condition is `TRUE` only if the value of the expression on the left of the `BETWEEN` keyword is in the inclusive range of the values of the two expressions on the right of the `BETWEEN` keyword.
- If the `NOT` keyword immediately precedes the `BETWEEN` keyword, the BETWEEN condition is `TRUE` only if the value of the expression on the left of the `BETWEEN` keyword is not in the inclusive range of the values of the two expressions on the right of the `BETWEEN` keyword.

Otherwise, the BETWEEN condition is `FALSE`.

Examples of BETWEEN conditions

The following examples illustrate BETWEEN conditions:

```
order_date BETWEEN '6/1/97' and '9/7/97'

zipcode NOT BETWEEN '94100' and '94199'

EXTEND(call_dtime, DAY TO DAY) BETWEEN
   (CURRENT - INTERVAL(7) DAY TO DAY) AND CURRENT

lead_time BETWEEN INTERVAL (1) DAY TO DAY
   AND INTERVAL (4) DAY TO DAY

unit_price BETWEEN loprice AND hiprice
```

IN Condition

The IN condition is satisfied when the expression to the left of the keyword IN is included in the list of items.

(explicit id unique_1057_Connect_42_incon003) unique_1057_Connect_42_incon003(explicit id unique_1057_Connect_42_incon004)
unique_1057_Connect_42_incon004(explicit id unique_1057_Connect_42_incon005) unique_1057_Connect_42_incon005(explicit
id unique_1057_Connect_42_incon006) unique_1057_Connect_42_incon006(explicit id unique_1057_Connect_42_incon007)
unique_1057_Connect_42_incon007(explicit id unique_1057_Connect_42_incon008) unique_1057_Connect_42_incon008(explicit id
unique_1057_Connect_42_incon009) unique_1057_Connect_42_incon009(explicit id unique_1057_Connect_42_incon010) unique_1057_Connect_42_incon010

IN Condition

" <Expression>[] "

" [NOT] "

" IN "

" { ({ | <Literal Number>[] | <Literal DATETIME>[] | <Quoted String>[] | <Literal INTERVAL>[] | USER |
CURRENT_USER | TODAY | CURRENT [<DATETIME Field Qualifier>[]] | SITENAME | DBSERVERNAME | <Literal Row>[] })
| { *collection_col* | (<Literal Collection>[]) | <Literal Collection>[] } } "

Element	Description	Restrictions	Syntax
collection_col	Name of a collection column that is used in an IN condition	The column must exist in the specified table	Identifier on page 1231

If you specify the NOT operator, the IN condition is `TRUE` when the expression is not in the list of items. NULL values do not satisfy the IN condition.

The following examples show some IN conditions:

```
WHERE state IN ('CA', 'WA', 'OR')
WHERE manu_code IN ('HRO', 'HSK')
WHERE user_id NOT IN (USER)
WHERE order_date NOT IN (TODAY)
```

In , the built-in TODAY function is evaluated at execution time. The built-in CURRENT function is evaluated when a cursor opens or when the query executes, if it is a singleton SELECT statement.

The built-in USER function is case sensitive; for example, it interprets **minnie** and **Minnie** as different values.

Using the IN operator with collection data types

You can use the IN operator to determine if an element is contained in a collection.

The collection can be a simple or nested collection. (In a *nested* collection type, the element type of the collection is also a collection type.) When you use IN to search for an element of a collection, the expression to the left or right of the IN keyword cannot contain a BYTE or TEXT data type.

Suppose you create the following table that contains two collection columns:

```
CREATE TABLE tab_coll
(
set_num SET(INT NOT NULL),
list_name LIST(SET(CHAR(10) NOT NULL) NOT NULL)
);
```

The following statement fragments show how you might use the IN operator for search conditions on the collection columns of the **tab_coll** table:

```
WHERE 5 IN set_num
WHERE 5.0::INT IN set_num
WHERE "5" NOT IN set_num
WHERE set_num IN ("SET{1,2,3}", "SET{7,8,9}")
WHERE "SET{'john', 'sally', 'bill'}" IN list_name
WHERE list_name IN ("LIST{""SET{'bill','usha'}""},
                    ""SET{'ann' 'moshi'}""}",
                "LIST{""SET{'bob','ramesh'}""},
                    ""SET{'bomani' 'ann'}""}")
```

In general, when you use the IN operator on a collection data type, the database server checks whether the value on the left of the IN operator is an element in the set of values on the right of the IN operator.

IS NULL and IS NOT NULL Conditions

The IS NULL condition is satisfied if the term that immediately precedes the IS keyword specifies one of the following undefined values:

- The name of a *column* that contains a null value.
- An *expression* that evaluates to null.

Conversely, if you use the IS NOT NULL operator, the condition is satisfied when the *column* contains a value that is not null, or when the *expression* that immediately precedes the IS NOT NULL keywords does not evaluate to null.

Suppose that you wish to perform an arithmetic computation on a column that can contain NULL values. You can create a table, insert values into the table, and then perform a query that uses a generic CASE expression that converts null values to `0` for the purpose of arithmetic calculations:

```
CREATE TABLE employee (emp_id INT, savings_in_401k INT, total_salary INT);

INSERT INTO employee VALUES(1, 5000, 40000);
INSERT INTO employee VALUES(2, 0, 40000);
INSERT INTO employee VALUES(3, NULL, 100000);

SELECT emp_id, savings_in_401k AS employer_match FROM employee WHERE
    CASE WHEN(savings_in_401k IS NULL) THEN 0
         ELSE savings_in_401k END * 0.06 > 0;
```

This example shows that by using IS NULL in the CASE expression, you can provide a value for the entries that otherwise are not computable because null is not a valid numeric value.

The IS NULL condition is satisfied if the column contains a null value or if the expression cannot be evaluated because it contains one or more null values. If you use the IS NOT NULL operator, the condition is satisfied when the operand is column value that is not null, or an expression that does not evaluate to null.

Trigger-Type Boolean Operator

The trigger-type Boolean operators of Informix® can test at runtime whether the currently executing triggered action was triggered by the specified type of DML event. These operators take no operands.

Trigger-Type Boolean Operator

"[{ DELETING | INSERTING | SELECTING | UPDATING }]"

These operators return TRUE ('t') if the triggering event of the currently executing trigger is the DML operation corresponding to the name of the operator, and they return FALSE ('f') otherwise. These operators are valid in IF statements, in CASE expressions, and in other contexts within an SPL trigger routines where a Boolean condition is valid.

For example, in the following statement fragment, the LET statement in the first THEN clause is executed only if the currently executing trigger was activated by an INSERT event, and the LET statement in the second THEN clause is executed only if the trigger was activated by a DELETE event:

```
IF (INSERTING = 't') THEN
   LET square = NEW.X * NEW.X
ELIF (DELETING = 't') THEN
   LET square = 0
```

The SELECTING, DELETING, INSERTING, and UPDATING operators are valid only in trigger UDRs that are invoked in the FOR EACH ROW triggered action of a trigger on a table, or (for the DELETING, INSERTING, and UPDATING operators) of an INSTEAD OF trigger on a view. An error is issued if you attempt to use a trigger-type Boolean operator in any other context.

If a trigger routine is invoked by a Delete, Insert, or Update trigger that the MERGE statement has activated,

- DELETING returns TRUE while MERGE is deleting a row from the target table.
- INSERTING returns TRUE while MERGE is inserting a row into the target table.
- UPDATING returns TRUE while MERGE is updating a row of the target table.

LIKE and MATCHES Condition

A LIKE or MATCHES condition tests for matching character strings.

The condition is `TRUE`, or satisfied, when either of the following tests is `TRUE`:

- The value of the column on the left matches the pattern that the quoted string specifies. You can use wildcard characters in the string. NULL values do not satisfy the condition.
- The value of the column on the left matches the pattern that the column on the right specifies. The value of the column on the right serves as the matching pattern in the condition.

If the quoted string includes literal characters that match any of the wildcard characters that the LIKE or MATCHES operator recognizes, the ESCAPE clause can define an ASCII character that you can include in the quoted string. When the column value on the left is compared to the quoted string, the next character that immediately follows this *escape character* is interpreted as a literal character, rather than as a wildcard, and the escape character is ignored. The LIKE and MATCHES operators recognize different wildcard characters. For more information about LIKE and MATCHES escape characters, see ESCAPE with LIKE on page 979 and ESCAPE with MATCHES on page 979 topics.

You can use the single quotation mark (`'`) only with the quoted string to match a literal single quotation mark; you cannot use the ESCAPE clause. You can use the single quotation mark character as the escape character in matching any other pattern if you write it as this: `''''`.



Important: Columns that you specify in LIKE or MATCHES conditions should be simple character data types, like CHAR, LVARCHAR, NCHAR, NVARCHAR, or VARCHAR. You cannot, for example, specify a complex data type, such as a ROW-type column, in a LIKE or MATCHES condition. (A ROW-type column is a column that is declared as a named or unnamed ROW type.) Similarly, the database server cannot evaluate a condition that uses LIKE or MATCHES with a simple or smart large object column, such as a CLOB column; a query that includes this condition fails with error -640.

NOT Operator

The NOT operator makes the search condition successful when the column on the left has a value that is not NULL and that does not match the pattern that the quoted string specifies.

For example, the following conditions exclude all rows that begin with the characters `Baxter` in the **lname** column:

```
WHERE lname NOT LIKE 'Baxter%'
WHERE lname NOT MATCHES 'Baxter*'
```

LIKE Operator

LIKE is the ANSI/ISO standard operator for comparing a column value to another column value, or to a quoted string.

The LIKE operator supports these wildcard characters in the quoted string.

Wildcard	Effect
%	Matches zero or more characters
_	Matches any single character

Besides `%` and `_`, LIKE supports a third wildcard character when both the DEFAULTESCCHAR configuration parameter and the DEFAULTESCCHAR session environment variable are not set:

Wildcard

	Effect
****	Removes the special significance of the next character (to match a literal `%` or `_` or `\` by specifying `\%` or `\_` or `\ \`)

Using the backslash (\ ) symbol as the default escape character (when DEFAULTESCCHAR is not set) is the Informix® extension to the ANSI/ISO-standard for SQL. You can specify backslash (\ ) symbol or some other ASCII character as the default escape character by setting the DEFAULTESCCHAR value to that character. For more information, see DEFAULTESCCHAR session environment option.

In an ANSI-compliant database, you can only use the LIKE escape character to escape a percent sign (%), an underscore (_), or the escape character itself.

The following condition tests the **description** column for the string `tennis`, alone or in a longer string, such as `tennis ball` or `table tennis paddle`:

```
WHERE description LIKE '%tennis%' ESCAPE '\'
```

The next example tests **description** for rows containing an underscore character. Here the backslash (\ ) escape character is necessary because underscore (_) is a wildcard character.

```
WHERE description LIKE '%\_%' ESCAPE '\'
```

The LIKE operator has an associated operator function called **like( )**. You can define a **like( )** function to handle your own user-defined data types. See also *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

MATCHES Operator

The MATCHES operator is the Informix® extension for comparing a column value to another column value, or to a quoted string.

The MATCHES operator supports these wildcard characters in the quoted string.

Wildcard	Effect
*	Matches any string of zero or more characters
?	Matches any single character
[. . .]	Matches any of the enclosed characters, including ranges, as in [`a-z`]. Characters within the brackets cannot be escaped.

^

As first character within the brackets, matches any character that is not listed. Thus, [^abc] matches any character except a, b, or c.

****

Removes the special significance of the next character (to match a literal \ or any other wildcard by specifying \ \ or* or \? and so forth)

The following condition tests for the string tennis, alone or within a longer string, such as tennis ball or table tennis paddle:

```
WHERE description MATCHES '*tennis*'
```

The following condition is TRUE for the names Frank and frank:

```
WHERE fname MATCHES '[Ff]rank'
```

The following condition is TRUE for any name that begins with either F or f:

```
WHERE fname MATCHES '[Ff]*'
```

The next condition is TRUE for any name that ends with the letters a, b, c, or d:

```
WHERE fname MATCHES '*[a-d]'
```

MATCHES has an associated **matches( )** operator function. You can define a **matches( )** function for your own user-defined data types. For more information, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

If **DB_LOCALE** or SET COLLATION specifies a nondefault locale supporting a localized collation, and you specify a range for the MATCHES operator using bracket ([. . .]) symbols, the database server uses the localized collating order, instead of code-set order, to interpret the range and to compare values that have CHAR, CHARACTER VARYING, LVARCHAR, NCHAR, NVARCHAR, and VARCHAR data types.

This behavior is an exception to the usual rule that only NCHAR and NVARCHAR data types can be compared in a localized collating order. For more information on the aspects of conditions that include the MATCHES or LIKE operators, see the *HCL® Informix® GLS User's Guide*.

In a NLSCASE INSENSITIVE database, comparison operations on NCHAR and NVARCHAR data disregard lettercase differences, so that the database server treats case variants among strings composed of same sequence letters as duplicates. All pairs of the following strings return TRUE as operands of the MATCHES operator:

```
'beta' 'Beta' 'BETA' 'bETa' 'betA' 'BetA'
```

For more information, see Duplicate rows in NLSCASE INSENSITIVE databases on page 699 and NCHAR and NVARCHAR expressions in case-insensitive databases.

ESCAPE with LIKE

The ESCAPE clause can specify an escape character that is different from the default escape character. The default escape character is set by the DEFAULTESCCHAR configuration parameter or the **DEFAULTESCCHAR** session environment option.

For example, if you specify `z` in the ESCAPE clause, then a quoted string operand that included `z_` is interpreted as including a literal underscore (_) character, rather than `_` as a wildcard. Similarly, `z%` is interpreted as a literal percent (`%`) sign, rather than % as a wildcard. Finally, the characters `zz` in a string would be interpreted as single literal `z`. The following statement retrieves rows from the **customer** table in which the **company** column includes a literal underscore character:

```
SELECT * FROM customer WHERE company LIKE '%z_%' ESCAPE 'z';
```

You can also use a host variable that contains a single character. The next statement uses a host variable to specify an escape character:

```
EXEC SQL BEGIN DECLARE SECTION;
   char escp='z';
   char fname[20];
EXEC SQL END DECLARE SECTION;
EXEC SQL select fname from customer
   into :fname where company like '%z_%' escape :escp;
```

ESCAPE with MATCHES

The ESCAPE clause can specify an escape character that is different from the default escape character. The default escape character is set by the DEFAULTESCCHAR configuration parameter or the **DEFAULTESCCHAR** session environment option.

Use this as you would the default escape character, the backslash, to include a question mark (?), an asterisk (*), a caret (^), or a left ([ ) or right ( ]) bracket as a literal character within the quoted string, to prevent them from being interpreted as special characters. If you choose to use `z` as the escape character, the characters `z?` in a string stand for a literal question mark (?). Similarly, the characters `z*` stand for a literal asterisk (*). Finally, the characters `zz` in the string stand for the single character `z`.

The following example retrieves rows from the **customer** table in which the value of the **company** column includes the question mark (?):

```
SELECT * FROM customer WHERE company MATCHES '*z?*' ESCAPE 'z';
```

Stand-Alone Condition

A stand-alone condition can be any expression that is not explicitly listed in the syntax for the comparison condition. Such an expression is valid as a condition only if it returns a BOOLEAN value. For example, the following example returns a value of the BOOLEAN data type:

```
funcname(x)
```

Condition with Subquery

Include a SELECT statement within a condition specifies a condition with subquery. You can use a subquery in a SELECT, INSERT, DELETE, or UPDATE statement to perform tasks like the following:

- Compare an expression to the result of the query.
- Determine if an expression is included in the results of the query.
- Ask whether the query selects any rows.

(explicit id unique_1270_Connect_42_csub001) unique_1270_Connect_42_csub001(explicit id unique_1270_Connect_42_csub002)

unique_1270_Connect_42_csub002(explicit id unique_1270_Connect_42_csub003) unique_1270_Connect_42_csub003

Condition with Subquery

" { <EXISTS Subquery> [] | <IN Subquery> [] | <ALL, ANY, SOME Subquery> [] } "

The subquery can depend on the current row that the outer SELECT statement is evaluating; in this case, the subquery is called a *correlated subquery*. (For a discussion of correlated subqueries and their impact on performance, see the *HCL® Informix® Guide to SQL: Tutorial*.)

The following sections describe subquery conditions and their syntax.

- For a discussion of types of subquery conditions in the context of the SELECT statement, see Using a Condition in the WHERE Clause on page 732.
- For a discussion of types of subquery conditions in the context of the INSERT statement, see Subset of SELECT Statement on page 590..
- For a discussion of types of subquery conditions in the context of the DELETE statement, see Subqueries in the WHERE Clause of DELETE on page 455..
- For a discussion of types of subquery conditions in the context of the UPDATE statement, see Subqueries in the WHERE Clause of UPDATE on page 879.

A subquery can return a single value, no value, or a set of values, depending on its context. If a subquery returns a value, it must select only a single column. If the subquery simply checks whether a row (or rows) exists, it can select any number of rows and columns.

A subquery cannot reference BYTE or TEXT columns, nor can it contain an ORDER BY clause. A subquery that specifies a table expression in the FROM clause, however, can include the ORDER BY clause.

A subquery and its outer DML statement operate on the same table object if the FROM clause of the subquery specifies the same table or view that the outer statement references in one of these clauses:

- in the FROM clause of the DELETE or SELECT statement
- in the INTO clause of the INSERT statement
- in the Table Options or Collection Derived Table specification of the UPDATE statement.

Subqueries that return more than one row and that operate on the same table or view as the enclosing DML statement are valid only in the WHERE clause of the DELETE or UPDATE statement. Even in this context, such subqueries return error -360 unless all of the following conditions are satisfied:

- The subquery does not reference any column name in its FROM list that is in a table not specified in the projection list
- The subquery is specified using the Condition with Subquery syntax.
- Any SPL routines within the subquery cannot reference the table that is being modified.

The following program fragment includes examples of conditions with subqueries in UPDATE and DELETE statements:

```
CREATE TABLE t1 ( a INT, a1 INT)
CREATE TABLE t2 ( b INT, b1 INT) ;
. . .
UPDATE t1 SET a = a + 10 WHERE EXISTS
   (SELECT a FROM t1 WHERE a > 1);
UPDATE t1 SET a = a + 10 WHERE a IN
   (SELECT a FROM t1, t2  WHERE a > b
    AND a IN
         (SELECT a FROM t1 WHERE a > 50 ) );
      DELETE FROM t1 WHERE EXISTS
   (SELECT a FROM t1);
```

For more information about subqueries in the DELETE statement, see Subqueries in the WHERE Clause of DELETE on page 455.

For more information about subqueries in the UPDATE statement, see Subqueries in the WHERE Clause of UPDATE on page 879 .

IN Subquery

An IN subquery condition is `TRUE` if the value of the expression matches one or more of the values from the subquery. (The subquery must return only one row, but it can return more than one column.) The keyword IN is equivalent to the =ANY specification. The keywords NOT IN are equivalent to the !=ALL specification. See the ALL, ANY, and SOME Subqueries on page 982.

(explicit id unique_1049_Connect_42_insub001) unique_1049_Connect_42_insub001

IN Subquery

“ <Expression>[] ”

“ [**NOT**] ”

“ **IN** ”

“ (*subquery*) ”

Element	Description	Restrictions	Syntax
subquery	Embedded query	Cannot contain the FIRST nor the ORDER BY clause	SELECT statement on page 690

The following example of an IN subquery finds the order numbers for orders that do not include baseball gloves (`stock_num` = `1`):

```
WHERE order_num NOT IN
   (SELECT order_num FROM items WHERE stock_num = 1)
```

Because the IN subquery tests for the presence of rows, duplicate rows in the subquery results do not affect the results of the main query. Therefore, the UNIQUE or DISTINCT keyword in the subquery has no effect on the query results, although not testing duplicates can improve query performance.

EXISTS Subquery condition

An EXISTS subquery condition evaluates to `TRUE` if the subquery returns a row. With an EXISTS subquery, one or more columns can be returned. The subquery always contains a reference to a column of the table in the main query. If you use an aggregate function in an EXISTS subquery that includes no HAVING clause, at least one row is always returned.

EXISTS Subquery

" [NOT] "

" EXISTS "

" (*subquery*) "

Element	Description	Restrictions	Syntax
subquery	Embedded query	Cannot contain the FIRST nor the ORDER BY clause	SELECT statement on page 690

The following example of a SELECT statement with an EXISTS subquery returns the stock number and manufacturer code for every item that has never been ordered (and is therefore not listed in the **items** table). You can appropriately use an EXISTS subquery in this SELECT statement because you use the subquery to test both **stock_num** and **manu_code** in **items**.

```
SELECT stock_num, manu_code FROM stock
   WHERE NOT EXISTS (SELECT stock_num, manu_code FROM items
      WHERE stock.stock_num = items.stock_num AND
      stock.manu_code = items.manu_code);
```

The preceding example works equally well if you use SELECT * in the subquery in place of the column names, because the existence of the entire row is tested; specific column values are not tested.

ALL, ANY, and SOME Subqueries

Use the ALL, ANY, and SOME keywords to specify what makes the condition `TRUE` or `FALSE`. A search condition that is `TRUE` when the ANY keyword is used might not be `TRUE` when the ALL keyword is used, and vice versa.

(explicit id unique_1061_Connect_42_aas001) unique_1061_Connect_42_aas001(explicit id unique_1061_Connect_42_aas002)

unique_1061_Connect_42_aas002

ALL, ANY, SOME Subquery

" <Expression>[] "

" <Relational Operator>[] "

"[{ALL | ANY | SOME}]"

"(*subquery*)"

Element	Description	Restrictions	Syntax
subquery	Embedded query	Cannot contain the FIRST or the ORDER BY clause	SELECT statement on page 690

Using the ALL Keyword

The ALL keyword specifies that the search condition is `TRUE` if the comparison is `TRUE` for every value that the subquery returns. If the subquery returns no value, the condition is `TRUE`.

In the following example, the first condition tests whether each **total_price** is greater than the total price of every item in order number 1023. The second condition uses the **MAX** aggregate function to produce the same results.

```
total_price > ALL (SELECT total_price FROM items
                WHERE order_num = 1023)

total_price > (SELECT MAX(total_price) FROM items
                WHERE order_num = 1023)
```

Using the NOT keyword with an ALL subquery tests whether an expression is not `TRUE` for at least one element that the subquery returns. For example, the following condition is `TRUE` when the expression **total_price** is not greater than all the selected values. That is, it is `TRUE` when **total_price** is not greater than the highest total price in order number 1023.

```
NOT total_price > ALL (SELECT total_price FROM items
                  WHERE order_num = 1023)
```

Using the ANY or SOME Keywords

The ANY keyword denotes that the search condition is `TRUE` if the comparison is `TRUE` for at least one of the values that is returned. If the subquery returns no value, the search condition is `FALSE`. The SOME keyword is a synonym for ANY.

The following conditions are `TRUE` when the total price is greater than the total price of at least one of the items in order number 1023. The first condition uses the ANY keyword; the second uses the MIN aggregate function:

```
total_price > ANY (SELECT total_price FROM items
               WHERE order_num = 1023)

total_price > (SELECT MIN(total_price) FROM items
               WHERE order_num = 1023)
```

Using the NOT keyword with an ANY subquery tests whether an expression is not `TRUE` for all elements that the subquery returns. For example, the following condition is `TRUE` when the expression **total_price** is not greater than any selected value. That is, it is `TRUE` when **total_price** is greater than none of the total prices in order number 1023.

```
NOT total_price > ANY (SELECT total_price FROM items
                          WHERE order_num = 1023)
```

Omitting the ANY, ALL, or SOME Keywords

You can omit the keywords ANY, ALL, or SOME in a subquery if you know that the subquery will return exactly one value. If you omit the ANY, ALL, or SOME keywords, and the subquery returns more than one value, you receive an error. The subquery in the following example returns only one row because it uses an aggregate function:

```
SELECT order_num FROM items
   WHERE stock_num = 9 AND quantity =
      (SELECT MAX(quantity) FROM items WHERE stock_num = 9);
```

NOT Operator

If you preface a condition with the keyword NOT, the test is `TRUE` only if the condition that NOT qualifies is `FALSE`. If the condition that NOT qualifies has a NULL or an UNKNOWN value, the NOT operator has no effect.

The following truth table shows the effect of NOT with 3–valued Boolean operands. Here T represents a `TRUE` condition, F represents a `FALSE` condition, and a question mark (`?`) represents an `UNKNOWN` condition. (An `UNKNOWN` value can occur when an operand is NULL).

NOT	
T	F
?	?
F	T

The left column shows the value of the operand of the NOT operator, and the right column shows the returned value after NOT is applied to the operand.

Conditions with AND or OR

You can combine simple conditions with the logical operators AND or OR to form complex conditions.

The following SELECT statements contain examples of complex conditions in their WHERE clauses:

```
SELECT customer_num, order_date FROM orders
   WHERE paid_date > '1/1/97' OR paid_date IS NULL;
SELECT order_num, total_price FROM items
   WHERE total_price > 200.00 AND manu_code LIKE 'H
SELECT lname, customer_num FROM customer
   WHERE zipcode BETWEEN '93500' AND '95700'
   OR state NOT IN ('CA', 'WA', 'OR');
```

The following truth tables show the effect of the AND and OR operators. The letter T represents a `TRUE` condition, F represents a `FALSE` condition, and the question mark (?) represents an `UNKNOWN` value. An `UNKNOWN` value can occur when part of an expression that uses a logical operator is NULL.

OR	T	?	F
T	T	T	T
?	T	?	?
F	T	?	F

AND	T	?	F
T	T	?	F
?	?	?	F
F	F	F	F

The marginal values at the left represent the first operand, and values in the top row represent the second operand. Values within each 3x3 matrix show the returned value after the operator is applied to operands of those values.

If the Boolean expression evaluates to `UNKNOWN`, the condition is not satisfied.

Consider the following example within a WHERE clause:

```
WHERE ship_charge/ship_weight < 5
   AND order_num = 1023
```

The row where **order_num** = 1023 is a row where **ship_weight** is NULL. Because **ship_weight** is NULL, **ship_charge**/**ship_weight** is also NULL; therefore, the truth value of **ship_charge**/**ship_weight** < 5 is `UNKNOWN`. Because **order_num** = 1023 is `TRUE`, the AND table states that the truth value of the entire condition is `UNKNOWN`. Consequently, that row is not chosen. If the condition used an OR in place of the AND, the condition would be `TRUE`.

Data Type

The Data Type segment specifies the data type of a column, of a component of a collection, of a field within a ROW type, of a routine parameter, or of a value returned by an expression or by a cast function. Use this segment whenever you see a reference to a data type in a syntax diagram.

Syntax

(explicit id unique_109_Connect_42_daty001) unique_109_Connect_42_daty001(explicit id unique_109_Connect_42_daty003)

unique_109_Connect_42_daty003(explicit id unique_109_Connect_42_daty004) unique_109_Connect_42_daty004(explicit id unique_109_Connect_42_daty005)

unique_109_Connect_42_daty005

Data Type

" { <Built-In Data Type> | { | <User-Defined Data Type> | <Complex Data Type> } } "

Usage

Sections that follow summarize these data types. For more information, see the chapter about data types in the *HCL® Informix® Guide to SQL: Reference*.

Built-In Data Types

Built-in data types are data types that are defined by the database server.

(explicit id unique_1407_Connect_42_bidt001) unique_1407_Connect_42_bidt001(explicit id unique_1407_Connect_42_bidt003)

unique_1407_Connect_42_bidt003(explicit id unique_1407_Connect_42_bidt004) unique_1407_Connect_42_bidt004(explicit id

unique_1407_Connect_42_bidt005) unique_1407_Connect_42_bidt005(explicit id unique_1407_Connect_42_bidt006) unique_1407_Connect_42_bidt006(explicit

id unique_1407_Connect_42_bidt007) unique_1407_Connect_42_bidt007(explicit id unique_1407_Connect_42_bidt008)

unique_1407_Connect_42_bidt008(explicit id unique_1407_Connect_42_bidt009) unique_1407_Connect_42_bidt009

Built-In Data Type

```
" { BOOLEAN [] | <BSON and JSON data types> [] | <Character Data Type> [] | <Numeric Data Type> [] | [] <Large-Object
Data Type> [] | <Time Data Type> [] | IDSSECURITYLABEL [] } "
```

These data types are built into the database server in the sense that the information and support functions required to interpret and transfer these data types is part of the database server software, which supports *character*, *numeric*, *large-object*, and *time* categories of built-in data types.

BOOLEAN and other built-in opaque data types

Informix® also supports the BOOLEAN data type, which is a *built-in opaque data type* that can store `true`, `false`, or NULL values. The symbol `t` represents a literal BOOLEAN `true` value, and `f` represents a literal BOOLEAN `false` value.

BOOLEAN, BSON, JSON, and LVARCHAR are the only built-in opaque data types that can be returned by cross-server distributed queries or by other cross-server distributed DML operations. Column values of other built-in opaque data types cannot be retrieved by a distributed query (nor modified by INSERT, DELETE, MERGE, or UPDATE operations on a remote database) unless all of the tables that the DML operation accesses are in databases of the local Informix® instance.

Similarly, in UDRs that perform distributed operations on databases of other Informix® instances, BOOLEAN, BSON, JSON, and LVARCHAR are the only built-in opaque types that are valid as a parameter or as the returned data type of the UDR, which must be defined in all participating databases.

Besides the BOOLEAN type, other built-in opaque data types of Informix® include BLOB, , BSON, JSON, CLOB, LVARCHAR, IFX_LO_SPEC, IFX_LO_STAT, INDEXKEYARRAY, POINTER, RTNPARAMTYPES, SELFUNCARGS, STAT, CLIENTBINVAL, and XID data types. These built-in opaque types are supported in the local database and in distributed operations across databases of the same server instance. The first five of these types are discussed in subsequent sections of this chapter.

Informix® also supports the built-in opaque data types LOLIST, IMPEXP, IMPEXPBIN, and SENDRECV. These types cannot, however, be accessed in a remote database by DML operations, nor returned from a remote database by a UDR, because these data types do not have the required support functions. For more information about the data types that Informix® supports in distributed transactions, see Data Types in Distributed Queries on page 700.

BSON and JSON built-in opaque data types

The BSON and JSON data types are built-in opaque types of Informix® that can be accessed and manipulated in local and distributed queries and in other DML operations. These are SQL data types that directly support relational database operations on data in BSON or JSON document store format. However, if you plan to query JSON and BSON data through the wire listener, you must create your database objects, such as collections and indexes, through the wire listener. You can use SQL statements to query JSON and BSON data whether you created your database objects through the wire listener or with SQL statements.

JSON and BSON documents contain one or more fields, which are similar to columns, and values for fields. A JSON or BSON column can contain multiple documents, each similar to a row in a relational database. The data values in JSON and BSON documents can be numbers, strings, or Boolean values. Documents can be nested within documents.

The database server validates JSON or BSON documents that you insert into the database. The JSON data type contains plain text. The BSON data type is a binary format of the JSON data type. You can create a table column of either the JSON or the BSON data type, however, the BSON data type is better suited for storing structured data in an Informix® database. The database server can operate on data in BSON columns, but can only insert and display data in JSON columns.

JSON and BSON documents up to 4 KB are stored in-row. Documents that are greater than 4 KB in size are stored in the sbspace that is associated with the table, or the default sbspace if the table does not have a designated sbspace. The maximum size of a JSON or BSON document is 32 KB. The maximum size of a JSON or BSON column is limited only by the operating system.

JSON data type on page 987
BSON data type on page 988
Supported SQL operations for BSON and JSON data types on page 989
Restrictions on BSON and JSON data types on page 990
Examples on page 990

JSON data type

The JSON data type is a plain text format for entering and displaying structured data. JSON is named after the JavaScript Object Notation (JSON), a data-interchange format that is based on the object-literal notation of JavaScript.

A JSON document has the format:

JSON document

{ *field*: { "*value* " <JSON document> <JSON array> } }

JSON array

" [{ "*value* " <JSON document> <JSON array> }] "

Element	Description
field	A string of Unicode characters that represents a field name.
value	A string of Unicode characters that represents a data value.

JSON documents require the following notation:

- Braces ({ and }) surround each JSON document. JSON documents can be nested.
- A comma (,) separates field-value pairs.
- Double quotation marks (") delimit character values. Quotation marks are not necessary around numeric values.
- Brackets ([and]) surround arrays of values. Arrays can contain nested documents or arrays.

The following JSON document has a nested document and an array:

```
{ "person" : {
     "givenname" : "Jim",
     "surname" : "Flynn",
     "age" : 29,
     "cars" : [ "dodge", "olds" ]
     }
}
```

BSON data type

The BSON data type is the binary representation of a JSON data type format for serializing JSON documents.

When you insert JSON documents through the wire listener with MongoDB API commands, a BSON column that is named **data** is created in the specified collection. The MongoDB API command also automatically adds an ObjectId field-value pair. When you insert JSON and BSON documents through SQL statements or Informix® utilities, the documents do not contain ObjectIds unless you explicitly include them.

To create a BSON column in a table with an SQL statement, use the standard CREATE TABLE statement and specify a BSON column. The only valid default value for a BSON column is NULL.

The BSON data type differs from standard SQL data types in that you must cast the data or use built-in BSON processing functions:

- To insert data that is in plain text with an SQL statement into a BSON column, you must cast the data to JSON. If you do not provide the cast to JSON, the database server treats the document as a string.
- To view BSON data in a readable format when you select data from a BSON column with an SQL statement, you must either cast the column to JSON or use a BSON value function in a function expression. BSON value functions convert field values in BSON columns to standard SQL data types. You can use dot notation to select data from specific fields within the BSON column and a JSON cast to return the results in a readable format.
- To update BSON data with an SQL statement, use the BSON_GET or the BSON_UPDATE function.

- When you include a JSON document as an expression in an SQL statement that operates on a BSON column, you must cast the document to JSON and then to BSON.
- To create an index on a BSON field, use the BSON_GET function and the USING BSON keywords as the index-key specification in the CREATE INDEX statement.

For more information, see BSON processing functions.

Supported SQL operations for BSON and JSON data types

The following table lists supported DDL operations on BSON and JSON data types.

Table 15. DDL operations on BSON and JSON data type

Operation	Supported for BSON data type	Supported for JSON data type
Create a table or a temporary table with one or more columns of the data type or a distinct type of the data type	Yes	Yes
Alter a table to add a column of the data type	Yes	Yes
Drop a column of the data type	Yes	Yes
Create a B-tree index on a column of the data type	No	No
Create an index or a functional index on a field	Yes	No
Create a basic text search index on a column of the data type	Yes	Yes
Truncate tables with columns of the data type	Yes	Yes
Fragment a table based on field values	Yes	Yes
Create a view based on columns of the data type and field values	Yes	No
Compress data in a column of the data type	Yes	Yes
Create a cast on the data type	Yes	Yes
Create triggers on fields	Yes	No
Include a column of the data type in a **TimeSeries** row data type	Yes	No

The following table lists supported DML operations on BSON and JSON data types.

Table 16. DML operations on BSON and JSON data

Operation	Supported for BSON data type	Supported for JSON data type
Cast data	Yes	Yes
Use a cursor to read data	Yes	Yes
Select contents of column of the data type	Yes	Yes
Select field values with dot notation or with BSON processing functions	Yes	No
Update data	Yes	No
Insert data	Yes	Yes
Find the size of a field value	Yes	No
Update statistics on a table with columns of the data type	Yes	Yes
Merge tables based on field values	Yes	No
Join tables based on field values	Yes	No
Load and unload data with the LOAD and UNLOAD statements, external tables, the onload and onunload utilities, or the dbschema, dbexport, and dbimport utilities	Yes	Yes
Replicate data to Remote stand-alone secondary servers	Yes	No
Shard data among Enterprise Replication servers	Yes	No

Restrictions on BSON and JSON data types

The following operations are not supported for BSON or JSON data types:

- Encryption of BSON or JSON columns
- Table-level restore of tables that have BSON or JSON columns
- Capture data through the Change Data Capture API

Examples

The examples use the following document:

```
{ "person" : {
     "givenname" : "Jim",
```

```
     "surname" : "Flynn",
     "age" : 29,
     "cars" : [ "dodge", "olds" ]
     }
}
```

Example: Create a table with a BSON column

The following statements create a table with a BSON column and insert a document:

```
CREATE DATABASE testdb WITH LOG;
CREATE TABLE IF NOT EXISTS bson_table(bson_col BSON);

INSERT INTO bson_table VALUES(
  '{person:{givenname:"Jim",surname:"Flynn",age:29,cars:["dodge","olds"]}}'::JSON);
```

The document is explicitly cast to JSON and then implicitly cast to BSON.

Example: Select the whole BSON column

The following statement returns the contents of the BSON column:

```
SELECT bson_col::JSON FROM bson_table;

(expression)
{"person":{"givenname":"Jim","surname":"Flynn","age":29,"cars":["dodge","olds"]}}
```

If the column is not cast to JSON format, the returned data is in an unreadable binary format.

Example: Select a field-value pair with dot notation

The following statement returns the value of the **surname** field:

```
SELECT bson_col.person.surname::JSON FROM bson_table;

  (expression)
  {surname:"Flynn"}
```

The dot notation uses the same format as *table_name.column_name*, but specifies a field within the column. The cast to JSON is necessary, and the returned value is a JSON document with the specified field-value pair.

Example: Select a field value with a BSON value function

The following statement returns the value of the **surname** field:

```
SELECT BSON_VALUE_LVARCHAR(bson_col, "person.surname") FROM bson_table;

   (expression)
   Flynn
```

The BSON_VALUE_LVARCHAR function converts the value of the **surname** field to an LVARCHAR data type. A cast to JSON is not needed, and the returned value is only the value of the specified field.

Example: Create an index on a BSON field

The following statement creates an index on the **surname** field in the BSON column:

```
CREATE INDEX idx2 ON bson_table(
   BSON_GET(bson_col, "person.surname")) USING BSON;
```

The BSON_GET function specifies that the **surname** field is indexed. The USING BSON keywords are necessary to specify that the index is created on a BSON column.

IDSSECURITYLABEL Data Type

In a table that is protected by a label-based security policy, the IDSSECURITYLABEL data type of Informix® stores a security label.

Only a user who holds the DBSECADM role can create, alter, or drop a column of this data type. This is a built-in DISTINCT OF VARCHAR(128) data type, but it is not classified as a character data type because its use is restricted to label-based access control. A table that has a security policy can have no more than one IDSSECURITYLABEL column, and a table that is associated with no security policy can have none.

The DBSECADM can use the GRANT statement to associate a specific security label with a user, and the REVOKE statement can cancel a security label that a user holds. For a given security policy, a user can have no more than one label that supports both read and write access, or no more than one label for write access and no more than one label for read access. For data protected by a security policy, but for which the user has been granted discretionary access privileges, the database server determines whether a specific user can access the data by comparing the security label of the data with the security label of the user, while also taking into consideration any exemptions to the security policy rules that the user holds.

For information on how to specify an IDSSECURITYLABEL value, see Security Label Support Functions on page 1092.

For a discussion of security policies, security components, security labels, and other concepts of label-based access control (LBAC), see the HCL Informix® Security Guide.

Numeric Data Types

Numeric data types enable the database server to store numbers such as integers and real numbers in a column.

(explicit id unique_1411_Connect_42_ndt001) unique_1411_Connect_42_ndt001(explicit id unique_1411_Connect_42_ndt002) unique_1411_Connect_42_ndt002

Numeric Data Type

“ { <Exact Numeric Data Type> [] | <Approximate Numeric Data Type> [] } ”

The values of numbers are stored either as *exact numeric* data types or as *approximate numeric* data types.

Large-Object Data Types

Large-object data types can store extremely large column values, such as images and documents, independently of the column.

Large-Object Data Type

"{{ TEXT | BYTE }[IN { TABLE | *blobspace* }]|{ BLOB | CLOB }}"

Element	Description	Restrictions	Syntax
blobspace	Name of an existing blobspace	Must exist	Identifier on page 1231

The large object data types can be classified in two categories:

- Simple large objects: TEXT and BYTE
- Smart large objects: CLOB and BLOB

Simple-large-object data types

A simple large object data type stores text or binary data in blobspaces.

These are the simple-large-object data types:

TEXT

Stores text data of up to 2^{31} bytes

BYTE

Stores any digitized data of up to 2^{31} bytes

Do not supply a BYTE value where TEXT is expected. No built-in cast supports BYTE to TEXT data type conversion.

Because of the maximum row size limit of 32,767 bytes, you cannot create a table with more than approximately 195 BYTE or TEXT columns. (This restriction also applies to all varying-length data types and ROW data types.)

For more information about the simple large object data types, see the *HCL® Informix® Guide to SQL: Reference*.

For information on how to create blobspaces, see your *HCL® Informix® Administrator's Guide*.

Storing BYTE and TEXT Data

A simple-large-object data type can store text or binary data in blobspaces or in tables. The database server can access a BYTE or TEXT value in one piece. When you specify a BYTE or TEXT data type, you can specify the location in which it is stored. You can store data with the table or in a separate blobspace.

If you are creating a named ROW data type that has a BYTE or TEXT field, you cannot use the IN clause to specify a separate storage space.

The following example shows how blobspaces and dbspaces are specified. The user creates the **resume** table. The data values are stored in the **employ** dbspace. The data in the **vita** column is stored with the table, but the data associated with the **photo** column is stored in a blobspace named **photo_space**.

```
CREATE TABLE resume
   (
   fname         CHAR(15),
   lname         CHAR(15),
   phone         CHAR(18),
   recd_date     DATETIME YEAR TO HOUR,
   contact_date  DATETIME YEAR TO HOUR,
   comments      VARCHAR(250, 100),
   vita          TEXT IN TABLE,
   photo         BYTE IN photo_space
   )
   IN employ;
```

Smart-large-object data types

A smart large object data type stores text or binary data in sbspaces.

The database server can provide random access to a smart large object value. That is, it can access any portion of the smart large object value. These data types are recoverable. The following list summarizes the smart large object data types that HCL Informix® supports.

BLOB

Stores binary data of up to 4 terabytes ($4*2^{40}$ bytes)

CLOB

Stores text data of up to 4 terabytes ($4*2^{40}$ bytes)

A smart large object is stored in a single sbspace. The SBSPACENAME configuration parameter specifies the system default sbspace in which smart large objects are created, unless you specify another storage area. For information about how the CREATE TABLE statement can specify nondefault storage locations and nondefault storage characteristics for BLOB or CLOB columns, see the description of the PUT Clause on page 352.

Both of these are built-in opaque data types. Like most opaque types, they cannot be accessed in a database of a non-local database server by a distributed query or by other DML operations, nor can they be returned from a database of another database server by a UDR. For information on accessing BLOB or CLOB values in other databases of the local server, however, see BOOLEAN and other built-in opaque data types on page 986.

Smart large object data types are not parallelizable. The PDQ feature of Dynamic Serve has no effect on operations that load or unload BLOB or CLOB values, or that process them in queries or in other DML operations.

For more information about the smart large object data types, see the *HCL® Informix® Guide to SQL: Reference*.

For information about how to create sbspaces, see your *HCL® Informix® Administrator's Guide*.

For information about the built-in functions that you can use to import, export, and copy smart large objects, see Smart-Large-Object Functions on page 1095 and the *HCL® Informix® Guide to SQL: Tutorial*.

Time Data Types

The time data types store calendar dates, points in time, and intervals of time.

(explicit id unique_1412_Connect_42_tdt001) unique_1412_Connect_42_tdt001(explicit id unique_1412_Connect_42_tdt002)

unique_1412_Connect_42_tdt002(explicit id unique_1412_Connect_42_tdt003) unique_1412_Connect_42_tdt003

Time Data Types

" { **DATE** | **INTERVAL** <INTERVAL Field Qualifier>[] | []**DATETIME** <DATETIME Field Qualifier>[] } "

The following table summarizes the built-in time data types.

Data Type

Description

DATE

Stores a date value as a Julian date in the range from January 1 of the year 1 up to December 31, 9*999*.

DATETIME

Stores a point-in-time date (*year*, *month*, *day*) and time-of-day (*hour*, *minute*, *second*, and *fraction* of second), in the range of years 1 to 9999. Also supports contiguous subsets of these time units.

INTERVAL

Stores spans of time, in *years* and/or *months*, or in smaller time units (*days*, *hours*, *minutes*, *seconds*, and/or *fractions* of second), with up to 9 digits of precision in the largest time unit, if this is not FRACTION. Also supports contiguous subsets of these time units.

For the order of precedence among the Informix® environment variables that can specify the display and data entry format of the built-in time data types, see the topic Precedence of DATE and DATETIME format specifications.

User-Defined Data Type

A user-defined data type is one that a user defines for the database server. Informix® supports two categories of user-defined data types, namely *distinct data types* and *opaque data types*. This is the declaration syntax for user-defined data types:

(explicit id unique_1408_Connect_42_uddt001) unique_1408_Connect_42_uddt001

User-Defined Data Type

" [<Owner Name>[] .] "

" { *opaque_type* | *distinct_type* } "

Element	Description	Restrictions	Syntax
distinct_type	Distinct data type with same structure as an existing data type	Must be unique among data type names in the database	Identifier on page 1231
opaque_type	Name of the opaque data type	Must be unique among data type names in the database	Identifier on page 1231

In this document, *user-defined data type* is usually abbreviated as UDT.

Distinct Data Types

A DISTINCT data type is a user-defined data type that is based on one of the following data types:

- a built-in type (including built-in opaque types)
- a user-defined opaque type
- a named ROW type
- an existing DISTINCT type.

The base type of a DISTINCT type cannot be any of the following data types:

- an unnamed ROW type
- a LIST, MULTISET, SET, or generic COLLECTION type.

The DISTINCT type inherits the length and the alignment of its base type in storage. Informix® automatically creates explicit casts between the DISTINCT type and its base type. To create a DISTINCT type, you must use the CREATE DISTINCT TYPE statement. (For more information, see CREATE DISTINCT TYPE statement on page 199.)

DISTINCT Types in Distributed Operations

Cross-server and cross-database operations can return DISTINCT types whose base types are built-in atomic data types, if the DISTINCT values are explicitly cast to a built-in data type. The base-type hierarchy and the casts must be identically defined in every participating database.

Restrictions on base types and on casts

Only a subset of built-in opaque types, however, can be the base type of a DISTINCT value that a distributed query returns. Complex DISTINCT types defined on a ROW base type can be returned only from the local database.

DISTINCT column values cannot be retrieved from another database of the same Informix® instance by a distributed query (nor modified by INSERT, DELETE, MERGE, or UPDATE cross-database distributed operations), nor by a function call, unless all of the following conditions are true:

- The DISTINCT type is defined on one of the following base types:
  - an non-opaque built-in atomic data type
  - a BOOLEAN, BSON, JSON, or LVARCHAR data type
  - a DISTINCT type defined on BOOLEAN, on BSON, on JSON, on LVARCHAR, or on a non-opaque built-in data type.

  (This condition also applies recursively to DISTINCT types of DISTINCT types, where the ultimate base type is BOOLEAN, BSON, JSON, or LVARCHAR, or a non-opaque built-in data type.)
- the DISTINCT type is explicitly cast to BOOLEAN, to BSON, to JSON, to LVARCHAR, or to a non-opaque built-in type
- the DISTINCT type, its type hierarchy, and its explicit cast to a built-in type are defined exactly the same way in all participating databases.

For DISTINCT data types in distributed operations, the data type hierarchy must have one of the following forms, which cannot vary across the participating databases:

Base-type hierarchy for Distributed DISTINCT Types

```
DISTINCT OF [ distinct_typeDISTINCT OF ] { BOOLEAN  | BSON  | JSON | LVARCHAR { ( 2048 ) | ( max ) }  | built-in_non-opaque_type }
```

Element	Description	Restrictions	Syntax
built-in _ non-opaque _type	Atomic built-in data type that is not opaque	Type cannot be complex, serial, BYTE, or TEXT	Data Type on page 985
distinct_t ype	DISTINCT type whose base type is another DISTINCT type	Root of this hierarchy must be a BSON, BOOLEAN, JSON, LVARCHAR, or an atomic *built-in non-opaque* data type	Data Type on page 985

Important:

The diagram above shows the generalized logical hierarchy of the base types for any DISTINCT data type. Using the DISTINCT OF keywords recursively, however, as in the diagram above, is not valid SQL syntax. The CREATE DISTINCT TYPE statement must specify exactly one base type for the new DISTINCT type. To create a hierarchy of DISTINCT data types, you must issue a separate CREATE DISTINCT TYPE statement for every DISTINCT type in the hierarchy. For the SQL syntax to define a new DISTINCT data type, see the topic CREATE DISTINCT TYPE statement on page 199.

A user-defined routine can return to the local database a DISTINCT data type from another database of the same Informix® instance only if all of the conditions listed above are true, and the UDR is defined in all of the participating databases.

The built-in DISTINCT type IDSSECURITYLABEL

The IDSSECURITYLABEL data type, which stores the security label in rows of protected tables, is a built-in DISTINCT type that satisfies this requirement, because its base type is the built-in VARCHAR(128) data type.

In queries that access a protected table in an Informix® database, the security label stored in the IDSSECURITYLABEL column prevents protected rows from being returned by the query, unless the user issuing the query holds sufficient LBAC credentials for the security policy of which that label is a component.

DISTINCT types in cross-server operations

The same rules that apply to DISTINCT data types in distributed operations across databases of the same Informix® instance also apply to DISTINCT data types in cross-server distributed operations on databases of different Informix® instances.

For additional information about the data types that Informix® supports in distributed operations, see Data Types in Distributed Queries on page 700.

Opaque Data Types

An opaque data type is a user-defined data type that can be used in the same way as a built-in data type. To create an opaque type, you must use the CREATE OPAQUE TYPE statement. Because an opaque type is encapsulated, you create support functions to access the individual components of an opaque type. The internal storage details of the type are hidden or opaque.

For more information about how to create an opaque data type and its support functions, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

Because of the maximum row size limit of 32,767 bytes, when you create a new table, no more than approximately 195 columns can be varying-length opaque or distinct user-defined data types. (The same restriction applies to BYTE, TEXT, VARCHAR, LVARCHAR, NVARCHAR, and ROW type columns. See ROW Data Types on page 999 for additional information about ROW data types.)

Complex Data Type

Complex data types are ROW types or COLLECTION types that you create from built-in types, opaque types, distinct types, or other complex types.

(explicit id unique_1409_Connect_42_cdty001) unique_1409_Connect_42_cdty001(explicit id unique_1409_Connect_42_cdty002)

unique_1409_Connect_42_cdty002

Complex Data Type

" { <Row Data Types>[] | <Collection Data Types>[] } "

A single complex data type can include multiple components. When you create a complex type, you define the components of the complex type. Unlike an opaque type, however, a complex type is not encapsulated. You can use SQL to access the individual components of a complex data type. The individual components of a complex data type are called *elements*.

Informix® supports the following categories of complex data types:

- ROW data types: Named ROW types and unnamed ROW types
- COLLECTION data types: SET, MULTISET, and LIST

The elements of a COLLECTION data type must all be of the same data type. You can use the keyword COLLECTION in SPL data type declarations to specify an untyped collection variable. NULL values are not supported in elements of COLLECTION data types.

The elements of a ROW data type can be of different data types, but the pattern of data types from the first to the last element cannot vary for a given ROW data type. NULL values are supported in elements of ROW data types, unless you specify otherwise in the data type declaration or in a constraint.

ROW Data Types

A ROW data type is a complex data type that can store multiple data values within its ordered set of one or more fields. Informix supports two categories of ROW data types: *named ROW types* that the CREATE ROW TYPE statement registers in the system catalog, and *unnamed ROW types* that ROW constructor expressions define.

You can use this syntax when you assign a ROW data type to a column or to an SPL variable.

Named and unnamed ROW data types

" { <Named ROW data type> <Unnamed ROW data type> } "

Named ROW data type

" [*owner* .] *row_type* "

Unnamed ROW data type

" ROW "

" [(*fielddata_type*)] "

Element	Description	Restrictions	Syntax
data_type	Data type of *field*	Any built-in type except BYTE or TEXT, or a UDT that has a support function for bit-hashing	Data Type on page 985
field	Name of a field within *row_type*	Must be unique among field names of the same ROW type	Identifier on page 1231

Element	Description	Restrictions	Syntax
owner	Authorization identifier of the owner of this ROW type	In an ANSI-compliant database, the combination *owner.row_type* must be unique among data types in the database	Owner name on page 1262
row_type	Named ROW type that CREATE ROW TYPE statement defined	ROW type must exist in the database	Identifier on page 1231; Data Type on page 985

Named ROW types

You can assign a named ROW type to a table, to a column, or to an SPL variable. A named ROW type must already exist in the database before you can use it in any of the following contexts:

- to create a typed table
- to define a column
- to define an SPL variable of a named ROW type.

A field in a named ROW type can be another existing named ROW type, as in this example:

```
CREATE ROW TYPE row_t ( w INT, y INT);
CREATE ROW TYPE rowspace_t ( u INT, v row_t, z DATE);
```

Each row of the named ROW type `rowspace_t` can stores three INT values and one DATE value, but two of those INT values are in the field of ROW type `row_t`.

To specify a named ROW data type in an ANSI-compliant database, you must qualify the *row_type* with its *owner* name, if you are not the owner of *row_type*.

A named ROW type can be the child of a supertype within a data type hierarchy, from which it inherits the field data types of its parent ROW type, or it can have no parent supertype. For the DDL syntax to create a new named ROW data type, see CREATE ROW TYPE statement on page 286.

Unnamed ROW types

An unnamed ROW data type is identified by its structure, which specifies fields that you create with its ROW constructor. For the syntax of unnamed ROW constructors, see Constructor Expressions on page 1049.

You can define a column or an SPL variable as an unnamed ROW data type.

If you omit the field list of the ROW constructor when the DEFINE statement of SPL declares an SPL variable as an unnamed ROW data type, the new variable has a generic ROW data type, to which the SPL routine can assign the field values of any ROW data type. For the syntax to declare SPL variables as ROW data types, see Subset of Complex Data Types on page 925.

Restrictions on ROW types

Columns in database tables cannot be generic ROW types.

An SPL variable declared as a generic ROW data type cannot return the result of an SQL routine. Before you can use a ROW type variable in an SPL routine, you must initialize the row variable with a LET statement or with a SELECT INTO statement.

Fields in a ROW data type cannot include TEXT or BYTE data types. Fields in ROW types also cannot be user-defined types (UDTs) that are not bit-hashable using the built-in hashing function of the database server.

Because of the maximum row size limit of 32,767 bytes, a single table cannot be created with more than approximately 195 ROW type columns.

Collection Data Types

This diagram shows the syntax to define a column or an SPL variable as a collection data type. (A table can include no more than 97 columns of collection data types.) For the syntax to specify values of collection elements, see Collection Constructors on page 1051.

Collection Data Type

" { COLLECTION | { SET | MULTISET | LIST } ({ data_type | (data_type NOT NULL) } NOT NULL) } "

Element	Description	Restrictions	Syntax
data_type	Data type of each of the elements of the collection	Can be any data type except BIGSERIAL, BYTE, SERIAL, or SERIAL8, or TEXT	Data Type on page 985

A SET is an unordered collection of elements, each of which has a unique value. Define a column as a SET data type when you want to store collections whose elements contain no duplicate values and have no associated order.

A MULTISET is an unordered collection of elements that can have duplicate values. You can define a column as a MULTISET collection type when you want to store collections whose elements might not be unique and have no specific order associated with them.

A LIST is an ordered collection of elements that can include duplicate elements. A LIST differs from a MULTISET in that each element in a LIST collection has an ordinal position in the collection. You can define a column as a LIST collection type when you want to store collections whose elements might not be unique but have a specific order associated with them.

The keyword COLLECTION can be used in SPL data type declarations to specify an untyped collection variable.

If you attempt to insert a collection that includes one or more duplicate values into a SET column, Informix® issues no error, but the duplicate values are ignored, and only the unique values are inserted.

Duplicate Elements in DML Operations on SET Columns

The SET data type does not allow duplicate element values in the same collection. If you attempt to insert duplicate elements into a SET data type, or to update a SET column or variable to a value that includes duplicate elements, the database server issues no error or warning when the INSERT or UPDATE statement executes, but only one of the duplicate elements is stored in the SET column or variable.

For example, suppose you create table **t3** with column **a** of the SET data type, and then you insert four rows, some of which include elements that have identical values:

```
  > CREATE TABLE t3(a SET(INT NOT NULL));

  Table created.

> INSERT INTO t3 VALUES( SET{10, 20, 30} );

  1 row(s) inserted.

> INSERT INTO t3 VALUES( SET{10, 20, 10});

  1 row(s) inserted.

> INSERT INTO t3 VALUES( SET{10, 10, 10});

  1 row(s) inserted.

> INSERT INTO t3 VALUES( SET{10,10,10});

  1 row(s) inserted.
```

When you look at the data values that were inserted into column **t3.a**, the four inserted rows include no duplicate element values:

```
> SELECT * FROM t3;

  a  SET{10         ,20         ,30         }
  a  SET{10         ,20         }
  a  SET{10         }
  a  SET{10         }

  4 row(s) retrieved.
```

In this example, Informix® silently discarded all but one instance of the duplicated elements from what the VALUES clause of the INSERT statement specified for each SET value.

Similar behavior occurs if the SET clause of the UPDATE statement includes duplicate elements within the same SET value. Declare collection columns of the MULTISET data type, rather than of the SET data type, if you want the database to store unordered sets that can include duplicate elements within the same collection

Defining the Element Type

The element type can be any data type except TEXT, BYTE, SERIAL, SERIAL8, or BIGSERIAL. You can nest collection types, using elements of a collection type.

Every element must be of the same type. For example, if the element type of a collection data type is INTEGER, every element must be of type INTEGER.

An exception to this restriction occurs if the database server determines that some elements of a collection of character strings are VARCHAR data types (whose length is limited to 255 or fewer bytes) but other elements are longer than 255 bytes. In this case, the collection constructor can assign a CHAR(*n*) data type to all elements, for *n* the length in bytes of the longest element. If this is undesirable, you can cast the collection to LVARCHAR, to prevent padding extra length in elements of the collection, as in this example:

```
LIST {'first character string longer than 255 bytes . . . ',
   'second character string longer than 255 bytes . . . ',
   'another character string'} ::LIST (LVARCHAR NOT NULL)
```

See Collection Constructors on page 1051 for additional information.

If the element type of a collection is an unnamed ROW type, the unnamed ROW type cannot contain fields that hold unnamed ROW types. That is, a collection cannot contain nested unnamed ROW data types.

The elements of a collection cannot be NULL. When you define a column as a collection data type, you must use the NOT NULL keywords to specify that the elements of the collection cannot be NULL.

Privileges on a collection data type are those of the database column. You cannot specify privileges on individual elements of a collection.

Expression

Data values in SQL statements must be represented as expressions. An *expression* is a specification, which can include operators, operands, and parentheses, that the database server can evaluate to one or more values, or to a reference to some database object.

Expressions can refer to values already in a table of the database, or to values derived from such data, but some expressions (such as TODAY, USER, or literal values) can return values that are independent of the database. You can use expressions to specify values in data-manipulation statements, to define fragmentation strategies, and in other contexts. Use the Expression segment whenever you see a reference to an expression in a syntax diagram.

In most contexts, however, you are restricted to expressions whose returned value is of some specific data type, or of a data type that can be converted by the database server to some required data type.

HCL® Informix® database servers support the following categories of expressions:

(explicit id unique_63_Connect_42_sqle006) unique_63_Connect_42_sqle006(explicit id unique_63_Connect_42_sqle007)

unique_63_Connect_42_sqle007(explicit id unique_63_Connect_42_sqle008) unique_63_Connect_42_sqle008(explicit id unique_63_Connect_42_sqle009)

unique_63_Connect_42_sqle009(explicit id unique_63_Connect_42_sqle010) unique_63_Connect_42_sqle010(explicit id unique_63_Connect_42_sqle011)

unique_63_Connect_42_sqle011(explicit id unique_63_Connect_42_sqle012) unique_63_Connect_42_sqle012(explicit id unique_63_Connect_42_sqle013)

unique_63_Connect_42_sqle013(explicit id unique_63_Connect_42_sqle014) unique_63_Connect_42_sqle014

SQL Expressions

"[{ - | + }]{ <Cast Expressions> | <Column Expressions> | <Conditional Expressions> | <Constant Expressions> | <Constructor Expressions> | <Function Expressions> | <Statement-Local Variable Expressions> | <Aggregate Expressions> | NULL | *variable* | *SPL_variable* | (<Expression>) }"

Binary Operators

"{ + | - | * | / | || }"

Element	Description	Restrictions	Syntax
SPL_variable	In an SPL routine, a variable that contains some expression type that the syntax diagram shows	Must conform to the rules for expressions of that type	Identifier on page 1231
variable	Host or program variable that contains some expression type that the syntax diagram shows	Must conform to the rules for expressions of that type	Language-specific rules for names

Usage

The following table lists the types of SQL expressions and describes what each type returns.

Table 17. SQL expressions

Expression Type	Description
Aggregate functions	Returns values from built-in or from user-defined aggregates
Arithmetic operators	Supports arithmetic operations on one (unary operators) or two (binary operators) numeric operands
Concatenation operator	Concatenates two string values
Cast operators	Explicit casts from one data type to another
Column expressions	Column values

Table 17. SQL expressions (continued)

Expression Type	Description
Conditional expressions	Returns values that depend on conditional tests
Constant expressions	Literal values in data manipulation (DML) statements
Constructor expressions	Dynamically creates values for complex data types
Function expressions	Returns values from built-in or user-defined functions
Statement-Local Variable expressions	References a statement-local variable (SLV) in the same SQL statement where it was declared

You can also use host variables or SPL variables as expressions.

List of Expressions

Each category of SQL expression includes many individual expressions.

The following table lists all the SQL expressions (and some operators) in alphabetical order. The columns in this table have the following meanings:

- **Name** gives the name of each expression.
- **Description** gives a short description of each expression.
- **Syntax** lists the page that shows the syntax of the expression.
- **Usage** shows the page that describes the usage of the expression.

Name	Description	Syntax	Usage
ABS function	Returns absolute value of a numeric argument	Algebraic Functions on page 1057	ABS Function on page 1058
ACOS function	Returns the arc cosine of a numeric argument	Trigonometric Functions on page 1108	ACOS Function on page 1110
ACOSH function	Returns the hyperbolic tangent of the specified numeric input	Trigonometric Functions on page 1108	ACOSH Function on page 1111
ADD_MONTHS function	Adds a specified number of months	Time Functions on page 1101	#unique_1443
Addition (+) operator	Returns the sum of two numeric operands	Expression on page 1003	Arithmetic Operators on page 1019
ASCII function	Returns the ASCII codepoint of the first character in its string argument	String-Manipulation Functions on page 1113	ASCII Function on page 1120

Name	Description	Syntax	Usage
ASIN function	Returns the arc sine of a numeric argument	Trigonometric Functions on page 1108	ASIN Function on page 1111
ASINH function	Returns the arc hyperbolic sine of the specified numeric input	Trigonometric Functions on page 1108	ASINH Function on page 1111
ATAN function	Returns the arc tangent of numeric argument	Trigonometric Functions on page 1108	ATAN Function on page 1111
ATAN2 function	Calculates the angular component of polar coordinate arguments	Trigonometric Functions on page 1108	ATAN2 Function on page 1112
ATANH function	Returns the hyperbolic tangent of the specified numeric input	Trigonometric Functions on page 1108	ATANH Function on page 1111
AVG function	Returns the mean of a set of numeric values	Aggregate Expressions on page 1154	AVG Function on page 1157
BITAND	Returns the bitwise AND of two arguments	#unique_1453	BITAND Function
BITANDNOT	Returns the bitwise ANDNOT of two arguments	#unique_1453	BITANDNOT Function
BITNOT	Returns the bitwise NOT of two arguments	#unique_1453	BITNOT Function
BITOR	Returns the bitwise OR of two arguments	#unique_1453	BITOR Function
BITXOR	Returns the bitwise XOR of two arguments	#unique_1453	BITXOR Function
CARDINALITY function	Returns the number of elements in a collection data type (SET, MULTISET, or LIST)	CARDINALITY Function on page 1070	CARDINALITY Function on page 1070
CASE expression	Returns a value that depends on which of several conditional tests evaluates to true	CASE Expressions on page 1032	CASE Expressions on page 1032
CAST expression	Converts an expression to a specified data type	CAST Expressions on page 1022	CAST Expressions on page 1022
Cast (**::**) operator	See "Double-colon (::) cast operator"	CAST Expressions on page 1022	CAST Expressions on page 1022

Name	Description	Syntax	Usage
CEIL function	Returns the smallest integer that is greater than or equal to its single argument	Algebraic Functions on page 1057	CEIL Function on page 1058
CHARACTER_ LENGTH function	See CHAR_LENGTH function. (In multibyte locales, this replaces the LENGTH function.)	Length functions on page 1090	CHAR_LENGTH Function on page 1092
CHAR_LENGTH function	Returns count of logical characters in a string argument	Length functions on page 1090	CHAR_LENGTH Function on page 1092
CHARINDEX function	Returns the location of a substring within a string	CHARINDEX function on page 1134	CHARINDEX function on page 1134
CHR	Returns a code point in the range `0` through `255` from the default code set	String-Manipulation Functions on page 1113	CHR Function on page 1130
COALESCE function	Returns the first non-NULL value from a series of expressions	COALESCE Function on page 1036	COALESCE Function on page 1036
Column expression	Column value from a table	Column Expressions on page 1025	Column Expressions on page 1025
CONCAT operator function	Concatenates the results of two expressions	String-Manipulation Functions on page 1113	CONCAT Function on page 1114
Concatenation (||) operator	Concatenates the results of two expressions	Expression on page 1003	Concatenation Operator on page 1021
Constant expression	Expression with a literal, fixed, or variant value	Constant Expressions on page 1040	Constant Expressions on page 1040
COS function	Returns the cosine of a radian expression	Trigonometric Functions on page 1108	COS Function on page 1109
COSH function	Returns the hyperbolic cosine of the argument, where the argument is an angle expressed in radians	Trigonometric Functions on page 1108	COSH function on page 1109
COUNT (as a set of functions)	Functions that return frequency counts Each form of the COUNT function is listed below.	Aggregate Expressions on page 1154	Overview of COUNT Functions on page 1158
COUNT (ALL *column*) function	See COUNT (*column*) function.	Aggregate Expressions on page 1154	COUNT column Function on page 1159

Name	Description	Syntax	Usage
COUNT (*column*) function	Returns the number of non-NULL values in a specified column	Aggregate Expressions on page 1154	COUNT column Function on page 1159
COUNT DISTINCT function	Returns the number of unique non-NULL values in a specified column	Aggregate Expressions on page 1154	COUNT DISTINCT and COUNT UNIQUE functions on page 1158
COUNT UNIQUE function	See COUNT DISTINCT function.	Aggregate Expressions on page 1154	COUNT DISTINCT and COUNT UNIQUE functions on page 1158
COUNT (*) function	Returns the cardinality of the set of rows that satisfy a query	Aggregate Expressions on page 1154	COUNT(*) function on page 1158
CUME_DIST function	Returns percentile rankings for each row in an OLAP partition	OLAP ranking function expressions on page 1173	CUME_DIST function on page 1179
CURRENT operator	Returns the current time as a DATETIME value that consists of the date and the time of day	Constant Expressions on page 1040	CURRENT Operator on page 1044
CURRENT_ROLE operator	Returns the currently enabled role of the user	Constant Expressions on page 1040	CURRENT_ROLE Operator on page 1042
CURRENT_USER operator	Returns the authorization identifier of the user. Synonym for USER operator.	Constant Expressions on page 1040	USER or CURRENT_USER Operator on page 1042
sequence.CURRVAL	Returns the current value of specified *sequence*	Constant Expressions on page 1040	Using CURRVAL on page 1048
DATE function	Converts a nondate argument to a DATE value	Time Functions on page 1101	DATE Function on page 1102
DAY function	Returns the day of the month as an integer	Time Functions on page 1101	DAY Function on page 1103
DBINFO (*option*)	Functions for retrieving database and session information. Each *option* is listed below.	DBINFO Function on page 1071	DBINFO Options on page 1072
DBINFO ('bigserial')	Returns most recently inserted BIGSERIAL value	DBINFO Function on page 1071	Using the 'serial8' and 'bigserial' options on page 1077

Name	Description	Syntax	Usage
DBINFO ('cdrsession')	Shows whether a DML operation is part of a replicated transaction	DBINFO Function on page 1071	Using the 'cdrsession' option on page 1076
DBINFO ('dbhostname')	Returns the host name of the database server to which a client application is connected	DBINFO Function on page 1071	Using the 'dbhostname' Option on page 1077
DBINFO ('dbname')	Returns the identifier of the database to which a client application is connected	DBINFO Function on page 1071	#unique_1487
DBINFO ('dbspace', *tblspace_number*)	Returns the name of a dbspace corresponding to a tblspace number	DBINFO Function on page 1071	Using the ('dbspace', tblspace_num) Option on page 1074
DBINFO ('get_tz')	Returns the time zone of the current session	DBINFO Function on page 1071	Using the 'get_tz' Option on page 1078
DBINFO ('serial8')	Returns most recently inserted SERIAL8 value	DBINFO Function on page 1071	Using the 'serial8' and 'bigserial' options on page 1077
DBINFO ('sessionid')	Returns the session ID of the current session	DBINFO Function on page 1071	Using the 'sessionid' Option on page 1075
DBINFO ('sqlca.sqlerrd1')	Returns the last serial value inserted in a table	DBINFO Function on page 1071	Using the 'sqlca.sqlerrd1' Option on page 1074
DBINFO ('sqlca.sqlerrd2')	Returns the number of rows processed by DML statements, and by EXECUTE PROCEDURE and EXECUTE FUNCTION statements	DBINFO Function on page 1071	Using the 'sqlca.sqlerrd2' Option on page 1075
DBINFO ('utc_current')	Returns the current Coordinated Universal Time (UTC) value.	DBINFO Function on page 1071	Using the 'utc_current' Option on page 1078
DBINFO ('utc_to_datetime', *expression*)	Returns the DATETIME value of an integer or column *expression* that specifies a UTC value.	DBINFO Function on page 1071	Using the 'utc_to_datetime' Option on page 1078
DBINFO ('version', *parameter*)	Returns all or part, as specified by the *parameter*, of the exact version of the database server	DBINFO Function on page 1071	#unique_1495

Name	Description	Syntax	Usage
	to which the client application is connected.		
DBSERVERNAME function	Returns the name of the database server	Constant Expressions on page 1040	DBSERVERNAME and SITENAME Operators on page 1043
DECODE function	Evaluates one or more expression pairs and compares the *when* expression in each pair with a specified value expression	DECODE Function on page 1038	DECODE Function on page 1038
DECRYPT_ BINARY function	Returns a plain-text BLOB data value after processing an encrypted BLOB argument	Encryption and decryption functions on page 1079	DECRYPT_BINARY Function on page 1086
DECRYPT_CHAR function	Returns a plain-text string or CLOB after processing an encrypted argument	Encryption and decryption functions on page 1079	DECRYPT_CHAR Function on page 1086
DEFAULT_ROLE operator	Returns the default role of the current user	Constant Expressions on page 1040	DEFAULT_ROLE Operator on page 1043
DEGREES function	Converts units of radians to degrees	Trigonometric Functions on page 1108	DEGREES function on page 1112
DELETING Boolean operator	Returns 't' if triggering event is a DELETE	Trigger-Type Boolean Operator on page 975	Trigger-Type Boolean Operator on page 975
DENSERANK function	Synonym for the DENSE_RANK function	OLAP ranking function expressions on page 1173	DENSE_RANK function on page 1178
DENSE_RANK function	Ranks each row in an OLAP partition, with no gaps in ranks	OLAP ranking function expressions on page 1173	DENSE_RANK function on page 1178
Division (/) operator	Returns the quotient of two numeric operands	Expression on page 1003	Arithmetic Operators on page 1019
Double-colon (::) cast operator	Converts the value of an expression to a specified data type	CAST Expressions on page 1022	CAST Expressions on page 1022
Double-pipe (||) concatenation operator	Returns a string that joins one string operand to another string operand	Expression on page 1003	Concatenation Operator on page 1021

Name	Description	Syntax	Usage
ENCRYPT_AES function	Returns an encrypted string or BLOB after processing a plain-text string, BLOB, or CLOB	Encryption and decryption functions on page 1079	#unique_1504
ENCRYPT_TDES function	Returns an encrypted string or BLOB after processing a plain-text string, BLOB, or CLOB	Encryption and decryption functions on page 1079	ENCRYPT_TDES Function on page 1087
EXP function	Returns the exponent of a numeric expression	Exponential and Logarithmic Functions on page 1088	EXP Function on page 1088
EXTEND function	Resets precision of DATETIME or DATE value	Time Functions on page 1101	EXTEND Function on page 1106
FILETOBLOB function	Creates a BLOB value from data stored in a specified operating-system file	Smart-Large-Object Functions on page 1095	FILETOBLOB and FILETOCLOB Functions on page 1096
FILETOCLOB function	Creates a CLOB value from data stored in a specified operating-system file	Smart-Large-Object Functions on page 1095	FILETOBLOB and FILETOCLOB Functions on page 1096
FIRST_VALUE function	Returns the value of a specified expression for the first row in each OLAP window partition	OLAP aggregation function expressions on page 1181	LAST_VALUE function on page 1182
FLOOR function	Returns the largest integer that is smaller than or equal to its single argument	Algebraic Functions on page 1057	FLOOR Function on page 1058
FORMAT_UNITS function	Returns a character string that specifies a number and abbreviated units of memory or of storage	FORMAT_UNITS Function on page 1146	FORMAT_UNITS Function on page 1146
GETHINT function	Returns a plain-text hint string after processing an encrypted data-string argument	Encryption and decryption functions on page 1079	GETHINT Function on page 1088
GREATEST function	Returns the maximum value in a set of values	Algebraic Functions on page 1057	GREATEST function on page 1058
HEX function	Returns the hexadecimal encoding of a base-10 integer argument	HEX Function on page 1090	HEX Function on page 1090
Host variable	See Variable.	Expression on page 1003	Expression on page 1003

Name	Description	Syntax	Usage
IFX_ALLOW_ NEWLINE function	Sets a newline session mode that allows or disallows newline characters in quoted strings	IFX_ALLOW_NEWLINE Function on page 1148	IFX_ALLOW_NEWLINE Function on page 1148
INITCAP function	Converts a string argument to a string in which only the initial letter of each word is uppercase	Case-Conversion Functions on page 1130	INITCAP Function on page 1132
INSERTING Boolean operator	Returns 't' if triggering event is an INSERT	Trigger-Type Boolean Operator on page 975	Trigger-Type Boolean Operator on page 975
INSTR function	Returns position of Nth occurrence of a substring within a string	INSTR function on page 1136	INSTR function on page 1136
LAG function	Returns an expression value for the row at a specified offset before the current row in an OLAP partition	OLAP ranking function expressions on page 1173	LAG and LEAD functions on page 1174
LAST_DAY function	Returns the date of the last day of the month that its argument specifies	Time Functions on page 1101	#unique_1522
LAST_VALUE function	Returns the value of a specified expression for the last row in an OLAP window partition	OLAP aggregation function expressions on page 1181	LAST_VALUE function on page 1182
LEAD function	Returns an expression value for the row at a specified offset after the current row in an OLAP partition	OLAP ranking function expressions on page 1173	LAG and LEAD functions on page 1174
LEAST function	Returns the minimum value in a set of values	Algebraic Functions on page 1057	LEAST function on page 1059
LEFT function	Returns the leftmost N characters of a string	LEFT function on page 1138	LEFT function on page 1138
LEN function	Synonym for the LENGTH function	Length functions on page 1090	LENGTH Function on page 1091
LENGTH function	Returns the number of bytes in a character column, not including trailing blank spaces	Length functions on page 1090	LENGTH Function on page 1091

Name	Description	Syntax	Usage
LIST collection constructor	Constructor for ordered collections that can contain duplicate values	Collection Constructors on page 1051	Collection Constructors on page 1051
Literal BOOLEAN	Literal representation of a BOOLEAN value	Constant Expressions on page 1040	Constant Expressions on page 1040
Literal collection	Represents elements in a collection data type	Constant Expressions on page 1040	Literal Collection on page 1049
Literal DATETIME	Represents a DATETIME value	Constant Expressions on page 1040	Literal DATETIME on page 1045
Literal INTERVAL	Represents an INTERVAL value	Constant Expressions on page 1040	Literal INTERVAL on page 1046
Literal number	Represents a numeric value	Constant Expressions on page 1040	Literal Number on page 1041
Literal opaque type	Represents an opaque data type	Constant Expressions on page 1040	Constant Expressions on page 1040
Literal row	Represents the elements in a ROW data type	Constant Expressions on page 1040	Literal Row on page 1049
LN	Returns the natural logarithm of a numeric argument	Exponential and Logarithmic Functions on page 1088	LN function on page 1089
LOCOPY function	Creates a copy of a smart large object	Smart-Large-Object Functions on page 1095	LOCOPY Function on page 1099
LOG10 function	Returns the base-10 logarithm of a numeric argument	Exponential and Logarithmic Functions on page 1088	LOG10 Function on page 1089
LOGN function	Returns the natural logarithm of a numeric argument	Exponential and Logarithmic Functions on page 1088	LOGN Function on page 1089
LOTOFILE function	Copies a BLOB or CLOB object to a file	Smart-Large-Object Functions on page 1095	LOTOFILE Function on page 1098
LOWER function	Converts uppercase letters to lowercase	Case-Conversion Functions on page 1130	LOWER Function on page 1132
LPAD function	Returns a string that is left-padded by a specified number of pad characters	String-Manipulation Functions on page 1113	LPAD Function on page 1127
LTRIM function	Removes specified leading pad characters from a string.	String-Manipulation Functions on page 1113	LTRIM Function on page 1123

Name	Description	Syntax	Usage
MAX function	Returns the largest in a specified set of values	Aggregate Expressions on page 1154	MAX Function on page 1162
MDY function	Returns a DATE value from integer arguments	Time Functions on page 1101	MDY Function on page 1106
MIN function	Returns the smallest in a specified set of values	Aggregate Expressions on page 1154	MIN Function on page 1162
MOD function	Returns the modulus (the integer-division remainder value) from two numeric arguments	Algebraic Functions on page 1057	MOD Function on page 1060
MONTH function	Returns the month value from a DATE or DATETIME argument	Time Functions on page 1101	MONTH Function on page 1103
MONTHS_ BETWEEN function	Returns the difference in months between two time arguments	Time Functions on page 1101	MONTHS_BETWEEN Function on page 1104
Multiplication (*) operator	Returns the product of two numeric operands	Expression on page 1003	Arithmetic Operators on page 1019
MULTISET collection constructor	Constructor for a non-ordered collection of elements that can contain duplicate value	Collection Constructors on page 1051	Collection Constructors on page 1051
NEXT_DAY function	Returns the earliest calendar date that satisfies both of two conditions	Time Functions on page 1101	#unique_1545
sequence.NEXTVAL	Increments value of the specified *sequence*	Constant Expressions on page 1040	Using NEXTVAL on page 1047
NTILE function	Classifies the rows in an OLAP partition into N ranked categories, called *tiles*, of similar cardinalities	OLAP ranking function expressions on page 1173	NTILE function on page 1180
NULL keyword	Unknown, missing, or logically undefined value	NULL Keyword on page 1053	NULL Keyword on page 1053
NULLIF function	Returns NULL if both arguments are equal	NULLIF Function on page 1038	NULLIF Function on page 1038
NVL function	Returns the value of a not-NULL argument, or a specified value if the argument is NULL	NVL Function on page 1037	NVL Function on page 1037

Name	Description	Syntax	Usage
NVL2 function	Returns the second argument when the first argument is not NULL	NVL2 Function on page 1089	NVL2 Function on page 1089
OCTET_LENGTH function	Returns the number of bytes in a character column, including any trailing blank spaces	Length functions on page 1090	OCTET_LENGTH Function on page 1092
PERCENT_RANK function	Returns a ranking value for each row in an OLAP window partition, normalized to a range from 0 to 1	OLAP ranking function expressions on page 1173	PERCENT_RANK function on page 1178
POW function	Raises a base value to a specified power	Algebraic Functions on page 1057	POW Function on page 1060
Power® function	Synonym for POW function	Algebraic Functions on page 1057	POW Function on page 1060
Procedure-call expression	See user-defined function.	User-Defined Functions on page 1149	User-Defined Functions on page 1149
Program variable	See variable.	Expression on page 1003	Expression on page 1003
QUARTER function	Returns the calendar quarter of a DATE or DATETIME value	Time Functions on page 1101	QUARTER Function on page 1103
Quoted string	Literal character string	Constant Expressions on page 1040	Quoted String on page 1041
RADIANS function	Converts units of degrees to radians	Trigonometric Functions on page 1108	RADIANS function on page 1113
RANGE function	Returns the range of a specified set of values	Aggregate Expressions on page 1154	RANGE Function on page 1163
RANK	Returns an ordinal number to rank each row in an OLAP window	OLAP ranking function expressions on page 1173	RANK function on page 1177
RATIOTOREPORT function	Synonym for the RATIO_TO_REPORT function	OLAP aggregation function expressions on page 1181	RATIO_TO_REPORT function on page 1183
RATIO_TO_REPORT function	Returns the fractional ratio of each row value to the sum for all rows in the same OLAP window partition	OLAP aggregation function expressions on page 1181	RATIO_TO_REPORT function on page 1183

Name	Description	Syntax	Usage
REPLACE function	Replaces specified characters in a source string	String-Manipulation Functions on page 1113	REPLACE Function on page 1126
REVERSE	Reverses the order of characters in a source string	String-Manipulation Functions on page 1113	REVERSE function on page 1125
RIGHT function	Returns the N rightmost characters from a source string	RIGHT function on page 1139	RIGHT function on page 1139
ROOT function	Returns a real, positive, Nth root value of a numeric argument	Algebraic Functions on page 1057	ROOT Function on page 1060
ROUND function	Returns the rounded value of an argument	Algebraic Functions on page 1057	ROUND Function on page 1061
ROW constructor	Constructor for a named ROW data type	Constructor Expressions on page 1049	ROW constructors on page 1050
ROWNUMBER function	Synonym for the ROW_NUMBER function	OLAP numbering function expression on page 1171	OLAP numbering function expression on page 1171
ROW_NUMBER function	Returns sequential integers for each row in an OLAP window partition	OLAP numbering function expression on page 1171	OLAP numbering function expression on page 1171
RPAD function	Returns a string right-padded by a specified number of pad characters	String-Manipulation Functions on page 1113	RPAD Function on page 1128
RTRIM function	Removes trailing blank pad characters from a string	String-Manipulation Functions on page 1113	RTRIM Function on page 1124
SECLABEL_BY_ COMP function	Returns the security label whose components are the arguments	Security Label Support Functions on page 1092	SECLABEL_BY_COMP Function on page 1093
SECLABEL_BY_ NAME function	Returns the security label whose identifier is the argument	Security Label Support Functions on page 1092	SECLABEL_BY_NAME Function on page 1093
SECLABEL_TO_ CHAR function	Returns the security label whose string format is the argument	Security Label Support Functions on page 1092	SECLABEL_TO_CHAR Function on page 1094
SELECTING Boolean operator	Returns 't' if triggering event is a SELECT	Trigger-Type Boolean Operator on page 975	Trigger-Type Boolean Operator on page 975
SET collection constructor	Constructor for an unordered collection of unique elements	Collection Constructors on page 1051	Collection Constructors on page 1051
SIGN function	Returns an indicator of the sign of the numeric argument	SIGN function on page 1095	SIGN function on page 1095

Name	Description	Syntax	Usage
SIN function	Returns the sine of a radians argument	Trigonometric Functions on page 1108	SIN Function on page 1109
SINH function	Returns the hyperbolic sine of a radians argument	Trigonometric Functions on page 1108	SINH function on page 1110
SITENAME function	See DBSERVERNAME function.	Constant Expressions on page 1040	DBSERVERNAME and SITENAME Operators on page 1043
SLV expression	A statement-local variable (SLV) whose scope is the SQL statement that declares it	Statement-Local Variable Declaration on page 1151	Statement-Local Variable Expressions on page 1153
SPACE function	Returns a string of N blank characters	String-Manipulation Functions on page 1113	SPACE function on page 1125
SPL routine expression	See "User-defined functions"	User-Defined Functions on page 1149	User-Defined Functions on page 1149
SPL variable	SPL variable that stores an expression	Expression on page 1003	Expression on page 1003
SQLCODE function	Returns **sqlca.sqlcode** value to an SPL UDR	SQLCODE Function (SPL) on page 1071	SQLCODE Function (SPL) on page 1071
SQRT function	Returns the square root of a numeric argument	Algebraic Functions on page 1057	SQRT Function on page 1061
STDEV function	Returns the standard deviation of a data set	Aggregate Expressions on page 1154	STDEV Function on page 1163
SUBSTR function	Returns a substring of a source string	SUBSTR function on page 1140	SUBSTR function on page 1140
SUBSTRB function	Returns a substring of a source string	SUBSTRB function on page 1141	SUBSTRB function on page 1141
SUBSTRING function	Returns a substring of a source string	SUBSTRING function on page 1143	SUBSTRING function on page 1143
SUBSTRING_INDEX function	Returns a substring that includes the Nth occurrence of a delimiter	SUBSTRING_INDEX function on page 1144	SUBSTRING_INDEX function on page 1144
Substring ([x, *y*]) operator	Returns a substring from a string operand	Column Expressions on page 1025	Using the Substring Operator on page 1030
Subtraction (-) operator	Returns the difference between two numbers	Expression on page 1003	Arithmetic Operators on page 1019

Name	Description	Syntax	Usage
SUM function	Returns the sum of a specified set of values	Aggregate Expressions on page 1154	SUM Function on page 1163
SYSDATE operator	Returns the current DATETIME value from the system clock.	Constant Expressions on page 1040	#unique_1587
TAN function	Returns the tangent of a radians expression	Trigonometric Functions on page 1108	TAN Function on page 1110
TANH function	Returns the hyperbolic tangent of a radians argument	Trigonometric Functions on page 1108	TANH Function on page 1110
TO_CHAR function	Converts a time or number to a character string	Time Functions on page 1101	#unique_1590
TO_DATE function	Converts a character string to a DATETIME value	Time Functions on page 1101	TO_DATE Function on page 1107
TO_NUMBER function	Converts a number or a character string to a DECIMAL value	TO_NUMBER Function on page 1107	TO_NUMBER Function on page 1107
TODAY operator	Returns the current system date	Constant Expressions on page 1040	TODAY Operator on page 1044
TRIM function	Drops blank pad characters from a character string argument	String-Manipulation Functions on page 1113	TRIM Function on page 1121
TRUNC function	Returns a truncated numeric or time value	Algebraic Functions on page 1057	TRUNC Function on page 1066
Unary minus (-) sign	Specifies a negative (< 0) numeric value	Expression on page 1003	Arithmetic Operators on page 1019
Unary plus (+) sign	Specifies a positive (> 0) numeric value .	Expression on page 1003	Arithmetic Operators on page 1019
UNITS operator	Convert an integer to an INTERVAL value	Constant Expressions on page 1040	UNITS Operator on page 1046
UPDATING Boolean operator	Returns 't' if triggering event is an UPDATE	Trigger-Type Boolean Operator on page 975	Trigger-Type Boolean Operator on page 975
UPPER function	Converts lowercase letters to uppercase	Case-Conversion Functions on page 1130	UPPER Function on page 1132
User-defined aggregate	Aggregate that a user defines (as opposed to a built-in aggregate)	User-Defined Aggregates on page 1166	User-Defined Aggregates on page 1166

Name	Description	Syntax	Usage
User-defined function	Function that a user writes (as opposed to a built-in function)	User-Defined Functions on page 1149	User-Defined Functions on page 1149
USER operator	Returns the authorization identifier of the current user	Constant Expressions on page 1040	USER or CURRENT_USER Operator on page 1042
Variable	Host or program variable that stores a value	Expression on page 1003	Expression on page 1003
VARIANCE function	Returns the variance for a set of numeric values	Aggregate Expressions on page 1154	VARIANCE Function on page 1164
WEEKDAY function	Returns an integer code for the day of the week	Time Functions on page 1101	WEEKDAY Function on page 1104
Window aggregate functions	Return aggregate results from OLAP window partitions	OLAP window expressions on page 1168	OLAP window aggregate functions on page 1185
YEAR function	Returns a 4-digit integer representing a year	Time Functions on page 1101	YEAR Function on page 1104
* symbol	See "Multiplication (*) operator"	Expression on page 1003	Arithmetic Operators on page 1019
+ symbol	See "Addition" and "Unary plus (+) sign"	Expression on page 1003	Arithmetic Operators on page 1019
- symbol	See "Subtraction" and "Unary minus (-) sign"	Expression on page 1003	Arithmetic Operators on page 1019
/ symbol	See "Division operator"	Expression on page 1003	Arithmetic Operators on page 1019
:: symbols	See "Double-colon (::) cast operator"	CAST Expressions on page 1022	CAST Expressions on page 1022
|| symbol	See "Double-pipe (||) concatenation operator"	Expression on page 1003	Concatenation Operator on page 1021
[*first*, *last*] symbols	See "Substring operator"	Column Expressions on page 1025	Using the Substring Operator on page 1030

Sections that follow describe the syntax and usage of each expression that appears in the preceding table.

Arithmetic Operators

Binary arithmetic operators can combine expressions that return numbers.

Arithmetic Operation	Arithmetic Operator	Operator Function	Arithmetic Operation	Arithmetic Operator	Operator Function
Addition	+	**plus( )**	Multiplication	*	**times( )**
Subtraction	–	**minus( )**	Division	/	**divide( )**

The following examples use binary arithmetic operators:

```
quantity * total_price
price * 2
COUNT(*) + 2
```

If you combine a DATETIME value with one or more INTERVAL values, all the fields of the INTERVAL value must be present in the DATETIME value; no implicit EXTEND function is performed. In addition, you cannot use YEAR to MONTH intervals with DAY to SECOND intervals. For additional information about binary arithmetic operators, see the *HCL® Informix® Guide to SQL: Reference*.

The binary arithmetic operators have associated operator functions, as the preceding table shows. Connecting two expressions with a binary operator is equivalent to invoking the associated operator function on the expressions. For example, the following two statements both select the product of the **total_price** column and 2. In the first statement, the * operator implicitly invokes the **times( )** function.

```
SELECT (total_price * 2) FROM items
   WHERE order_num = 1001;
SELECT times(total_price, 2) FROM items
   WHERE order_num = 1001;
```

You cannot use arithmetic operators to combine expressions that use aggregate functions with column expressions.

The database server provides the operator functions associated with the relational operators for all built-in data types. You can define new versions of these operator functions to handle your own user-defined data types.

For more information, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

The database server also supports the following unary arithmetic operators.

Sign of Number	Unary Arithmetic Operator	Operator Function
Positive	+	**positive( )**
Negative	–	**negate( )**

The unary arithmetic operators have the associated operator functions that the preceding table shows. You can define new versions of these functions to handle your own user-defined data types. For more information on this topic, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

If any value that participates in an arithmetic expression is NULL, the value of the entire expression is NULL, as the following example shows:

```
SELECT order_num, ship_charge/ship_weight FROM orders
   WHERE order_num = 1023;
```

If either **ship_charge** or **ship_weight** is NULL, the value returned for the expression **ship_charge**/**ship_weight** is also NULL. If the NULL expression **ship_charge**/**ship_weight** is used in a condition, its truth value cannot be TRUE, and the condition is not satisfied (unless the NULL expression is an operand of the **IS NULL** operator).

Concatenation Operator

The concatenation operator is a binary operator, whose syntax is shown in the general diagram for an SQL Expression on page 1003. You can use the concatenation operator (||) to concatenate two expressions that evaluate to character data types or to numeric data types. These examples show some possible concatenated expression combinations.

- The first example concatenates the **zipcode** column to the first three letters of the **lname** column.
- The second example concatenates the suffix **.dbg** to the contents of a host variable called **file_variable**.
- The third example concatenates the value that the **TODAY** operator returns to the string `Date`.

```
lname[1,3] || zipcode

:file_variable || '.dbg'

'Date:' || TODAY
```

You cannot use the concatenation operator in the following embedded-language statements:

- ALLOCATE COLLECTION
- ALLOCATE DESCRIPTOR
- ALLOCATE ROW
- CREATE FUNCTION FROM
- CREATE PROCEDURE FROM
- CREATE ROUTINE FROM
- DEALLOCATE COLLECTION
- DEALLOCATE DESCRIPTOR
- DEALLOCATE ROW DESCRIBE
- DESCRIBE INPUT
- EXECUTE
- FLUSH
- GET DESCRIPTOR
- GET DIAGNOSTICS
- PUT
- SET AUTOFREE
- SET CONNECTION
- SET DESCRIPTOR
- WHENEVER

Except as noted for the DECLARE and PREPARE statement, routines written in external languages, such as the language, cannot use the concatenation operator in the following dynamic SQL statements:

- CLOSE
- DECLARE
- EXECUTE IMMEDIATE
- FETCH
- FREE
- OPEN
- PREPARE

Although input parameters of the DECLARE statement, such as a *cursor_id* specification, cannot be expressions that include the concatenation operator, routines can use this operator in a SELECT, INSERT, EXECUTE FUNCTION, or EXECUTE PROCEDURE statement within the DECLARE statement.

routines can use the concatenation operator in the text of the SQL statement or statements that you pass to the PREPARE statement.

In SPL routines, you can include the concatenation operator in an expression that specifies the text of the SQL statement that you pass to the EXECUTE IMMEDIATE statement or to the PREPARE statement, even if the calling context of the SPL routine is routines.

You cannot use the concatenation operator directly with user-defined data types, with complex or large-object data types, nor with operands that are not built-in character or number data types. You must explicitly cast UDTs or other unsupported data types to a built-in character or numeric data type before you can pass the result to the concatenation operator.

The data type of the result of a concatenation operation depends of the data types of the operands and on the length of the resulting string, using the return type promotion rules that the section Return Types from the CONCAT Function on page 1115 describes.

The concatenation operator (`||`) has an associated operator function called **CONCAT**. The **CONCAT** function cannot be overloaded.

When you define a text-based UDT, you can define a **CONCAT** function to concatenate objects of that user-defined data type. For more information, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

CAST Expressions

Use the `CAST (... AS ...)` keywords or the double-colon (`::`) cast operator to convert the data type of the value of an *expression* to some other *target data type*.

If an SQL statement incudes the cast operator or the `CAST (... AS ...)` keywords, the database server examines the **syscasts** system catalog table for an existing cast corresponding to the data types of the *expression* value and the target data types that the Cast Expression specifies. If no built-in cast, explicit cast, or implicit cast registered in the system catalog can perform the specified conversion, the SQL statement returns an error.

(explicit id unique_1434_Connect_42_caex002) unique_1434_Connect_42_caex002

Cast Expressions

" { CAST (*expression* [::*target_type*] AS *target_type*) | *expression* ::*target_type* } "

Element	Description	Restrictions	Syntax
expression	Expression whose data type will be replaced by a *target_type*	Must evaluate to an atomic, or JSON document, or large object, or ROW data type	Expression on page 1003
target_type	Data type replacing the type returned by *expression* (or by an intermediate cast)	See Rules for the Target Data Type on page 1024	Data Type on page 985

Usage

A non-recursive cast expression converts a data value from its current data type directly to a specified target data type.

To execute an SQL statement that incudes the cast operator or the `CAST (... AS ...)` keywords, the database server searches the **syscasts** system catalog table for an existing cast corresponding to

- the data type of the *expression* value in the **syscasts.argument_type** column,
- and the *target data type* that the Cast Expression specifies in the **syscasts.result_type** column.

If no built-in cast, explicit cast, or implicit cast registered in the **syscasts** table matches these criteria for performing the specified conversion, the database server returns an error.

In this case, you might consider whether a recursive CAST expression is required, using intermediate data types. You might also consider whether you can use the CREATE CAST statement on page 188 to define and register in the system catalog an appropriate new explicit or implicit cast for data-type conversion to the target type.

For examples of explicit cast expressions for target types of non-opaque, user-defined, and large object data types, see Examples of Cast Expressions on page 1024.

Recursive CAST expressions

As the syntax diagram indicates, however, the `::target_type` syntax can be used recursively, so that the value of the original source expression is cast to one or more successive intermediate data types before it is cast to the last target type.

For example, the following expression casts a DATETIME (YEAR TO DAY) value to MONEY(16,2), using DATE, INTEGER, and DECIMAL as intermediate data types:

```
CAST (CURRENT::DATE::INTEGER::DECIMAL AS MONEY(16,2))
```

The next CAST expression, without the `CASE` or `AS` keywords, is logically equivalent:

```
CURRENT::DATE::INTEGER::DECIMAL::MONEY(16,2)
```

Tip: Use caution before interpreting this syntax example, which was designed to illustrate multiple casts within a single CAST expression, as validation of "Time is money" economic theories.

Rules for the Target Data Type

The following rules restrict the *target data type* in cast expressions:

- The target data type must be either a built-in type, a user-defined type, or a named row type in the database.
- The target data type cannot be an unnamed row or a collection type.
- The target data type can be a BLOB data type under the following conditions:
  - The source expression (the expression to be cast to another data type) is a BYTE data type.
  - The source expression is a user-defined type and the user has defined a cast from the user-defined type to the BLOB type.
- The target data type can be a CLOB type under these conditions:
  - The source expression is a TEXT data type.
  - The source expression is a user-defined type and the user has defined a cast from the user-defined type to the CLOB type.
- You cannot cast a BLOB data type to a BYTE data type.
- You cannot cast a CLOB data type to a TEXT data type.
- An explicit or implicit cast must exist that can convert the data type of the source expression to the target data type.

Examples of Cast Expressions

The following examples show two different ways to convert the sum of *x* and *y* to a user-defined data type, **user_type**. The two methods produce identical results. Both require the existence of an explicit or implicit cast from the type returned by `(x + y)` to the user-defined type:

```
CAST ((x + y) AS user_type)
(x + y)::user_type
```

The following examples show two different ways of finding the integer equivalent of the expression **expr**. Both require the existence of an implicit or explicit cast from the data type of **expr** to the INTEGER data type:

```
CAST (expr AS INTEGER)
expr::INTEGER
```

In the following example, the user casts a BYTE column to the BLOB type and copies the BLOB data to an operating-system file:

```
SELECT LOTOFILE(mybytecol::blob, 'fname', 'client')
   FROM mytab
   WHERE pkey = 12345;
```

In the following example, the user casts a TEXT column to a CLOB value and then updates a CLOB column in the same table to have the CLOB value derived from the TEXT column:

```
UPDATE newtab SET myclobcol = mytextcol::clob;
```

The Keyword NULL in Cast Expressions

Cast expressions can appear in the projection list, including expressions of the form NULL**::***datatype*, where *datatype* is any data type known to the database:

```
SELECT newtable.col0, null::int FROM newtable;
```

The keyword NULL has a global scope of reference within expressions. In SQL, the keyword NULL is the only syntactic mechanism for accessing a NULL value. Any attempt to redefine or restrict the global scope of the keyword NULL (for example, declaring an SPL variable called **null**) disables any cast expression that involves a NULL value. Make sure that the keyword NULL receives its global scope in all expression contexts.

Column Expressions

A *column expression* specifies a data value in a column in the database, or a substring of the value, or a field within a ROW-type column, or a field in a BSON column.

(explicit id unique_356_Connect_42_cexp003) unique_356_Connect_42_cexp003(explicit id unique_356_Connect_42_cexp004)

unique_356_Connect_42_cexp004(explicit id unique_356_Connect_42_cexp005) unique_356_Connect_42_cexp005

Column Expressions

" { { *table* . | *view* . | *synonym* . | *alias* . } { *column* [[[*first* , *last*]] | [**ROWID** | *row_column* [{ . * | [ ] . *row_field_name* }] }] | *row_col_expr* [{ . * | [ ] . *row_field_name* }] | *BSON_col_expr* [{ . * | [ ] . { *BSON_field_name* | *BSON_array_position* } }] } "

Element	Description	Restrictions	Syntax
alias	Temporary alternative name for a table or view, declared in the FROM clause of a query	Must return a string. Restrictions depend on the clause of the SELECT statement in which *alias* occurs	Identifier on page 1231
BSON_array_position	A positive integer that represents the position of a value in an array, starting with 0 for the first value.	Must be preceded by all ancestor field names.	
BSON_col_exp	Expression that returns a BSON column name	Must be of type BSON.	Expression on page 1003
BSON_field_name	BSON field name	Must be a literal BSON field name. Can be a multilevel field	Using Dot Notation on page 1028

Element	Description	Restrictions	Syntax
		identifier, up to 32 levels. All ancestor field names must be included.	
column	Name of a column	Restrictions depend on the SQL statement where *column* occurs	Identifier on page 1231
first , *last*	Integers indicating positions of first and last characters within *column*	The *column* must be of type CHAR, VARCHAR, NCHAR, NVARCHAR, BYTE, or TEXT, and 0 < *first* = *last*	Literal Number on page 1198
row_col_expr	Expression that returns ROW-type values	Must return a ROW data type	Expression on page 1003
row_column	Name of a ROW-type column	Must be a named ROW data type or an unnamed ROW data type	Identifier on page 1231
row_field_name	Name of a ROW field in the ROW column or ROW-column expression	Must be a member of the row that *row-column name* or *row_col_expr* or*row_field name* (for nested rows) specifies	Identifier on page 1231
synonym , *table*, *view*	Table, view, or synonym (for the table or view) that contains *column*	Synonym and the table or view to which it points must exist	Database Object Name on page 1225

You qualify a column name with a table name or alias to distinguish between columns that have the same name but are in different tables. You use dot notation to select a field from a ROW or a BSON column.

The following examples illustrate some formats of column expressions:

- Unqualified column name: `company`
- Column name that is qualified by table name: `items.price`
- Substring of a character-type column: `catalog_advert [1,15]`
- Multilevel BSON field identifier: `bson_col.person.cars.1`

In syntax contexts that support a default value for a column, you can omit the default column expression.

Example

Example: Qualify columns with table names

The following statement, two column expressions qualify the column name, **customer_num**, with the table names **customer** and **orders**:

```
SELECT * FROM customer, orders
   WHERE customer.customer_num = orders.customer_num;
```

Example

Example: Qualify columns with table aliases

The following statement includes column expressions that qualify column name, **customer_num**, with the table aliases **c** and **o**:

```
SELECT * FROM customer c, orders o
   WHERE c.customer_num = o.customer_num;
```

Example

Example: Select a field from a ROW column

The following statements creates a table with a ROW column that is named **rect** and that has four fields:

```
CREATE TABLE rectangles
(
   area float,
   rect ROW(x int, y int, length float, width float)
);
```

The following statement uses dot notation to access the **length** field of the **rect** column:

```
SELECT rect.length FROM rectangles
   WHERE area = 64;
```

Example

Example: Select a multilevel field from a BSON column

The following statements create and populate a BSON column that is named **bson_col**:

```
CREATE DATABASE testdb WITH LOG;
CREATE TABLE IF NOT EXISTS bson_table(bson_col BSON);

INSERT INTO bson_table VALUES(
   '{person:{givenname:"Jim",surname:"Flynn",age:29,cars:["dodge","olds"]}}'
   ::JSON::BSON);
```

Nested in the **person** field is an array that is named **cars**, which has two values. The values in arrays do not have field names. You specify array values with numbers that represent the position in the array, starting with 0 for the first value.

The following statement selects the second value in the **cars** array that is nested in the **person** field:

```
SELECT bson_col.person.cars.1::JSON FROM bson_table;

  (expression)
  {cars:"olds"}
```

Using Dot Notation

Dot notation (sometimes called the *membership operator*) qualifies an SQL identifier with another SQL identifier of which it is a component. You separate the identifiers with the period (.) symbol.

Column projections qualify a column name with the following SQL identifiers:

- Table name: *table_name.column_name*
- View name: *view_name.column_name*
- Synonym name: *syn_name.column_name*

Field projections directly access fields in a ROW or a BSON column: *column_name.field_name*. BSON documents can have a hierarchical structure. You can specify multiple fields and positions in arrays, each separated by a period. A field name or array position must be preceded by all its ancestor field names.

Selecting All Fields of a ROW Column with Asterisk Notation

If you want to select all fields of a column that has a ROW type, you can specify the column name without using dot notation. For example, you can select all fields of the **rect** column as follows:

```
SELECT rect FROM rectangles
   WHERE area = 64;
```

You can also use asterisk (*) notation to project all the fields of a column that has a ROW data type. For example, if you want to use asterisk notation to select all fields of the **rect** column, you can enter the following statement:

```
SELECT rect.* FROM rectangles
   WHERE area = 64;
```

Asterisk notation is easier than specifying each field of the **rect** column individually:

```
SELECT rect.x, rect.y, rect.length, rect.width
   FROM rectangles
      WHERE area = 64;
```

Asterisk notation for ROW fields is valid in the projection list of a SELECT statement. It can specify all fields of a ROW-type column or the data that a ROW-column expression returns.

Asterisk notation is not necessary with ROW-type columns, because you can specify the column name alone to project all of its fields. Asterisk notation is quite helpful, however, with ROW-type expressions such as subqueries and user-defined functions that return ROW-type values. For more information, see Using Dot Notation with Row-Type Expressions on page 1029.

You can use asterisk notation with columns and expressions of ROW data types in the projection list of a SELECT statement only. You cannot use asterisk notation with columns and expressions of ROW type in any other clause of a SELECT statement.

Selecting Nested Fields

When the ROW type that defines a column itself contains other ROW types, the column contains nested fields. Use dot notation to access these nested fields within a column.

For example, assume that the **address** column of the **employee** table contains the fields: **street**, **city**, **state**, and **zip**. In addition, the **zip** field contains the nested fields: **z_code** and **z_suffix**. A query on the **zip** field returns values for the **z_code** and **z_suffix** fields. You can specify, however, that a query returns only specific nested fields. The following example shows how to use dot notation to construct a SELECT statement that returns rows for the **z_code** field of the **address** column only:

```
SELECT address.zip.z_code
   FROM employee;
```

Rules of Precedence

The database server uses the following precedence rules to interpret dot notation:

1. schema *name_a* . table *name_b* . column *name_c* . field *name_d*
2. table *name_a* . column *name_b* . field *name_c* . field *name_d*
3. column *name_a* . field *name_b* . field *name_c* . field *name_d*

When the meaning of an identifier is ambiguous, the database server uses precedence rules to determine which database object the identifier specifies. Consider the following two tables:

```
CREATE TABLE b (c ROW(d INTEGER, e CHAR(2));
CREATE TABLE c (d INTEGER);
```

In the following SELECT statement, the expression **c.d** references column **d** of table **c** (rather than field **d** of column **c** in table **b**) because a table identifier has a higher precedence than a column identifier:

```
SELECT *
   FROM b,c
      WHERE c.d = 10;
```

For more information about precedence rules and how to use dot notation with ROW columns, see the *HCL® Informix® Guide to SQL: Tutorial*.

Using Dot Notation with Row-Type Expressions

Besides specifying a column of a ROW data type, you can also use dot notation with any expression that evaluates to a ROW type. In an INSERT statement, for example, you can use dot notation in a subquery that returns a single row of values. Assume that you created a ROW type named **row_t**:

```
CREATE ROW TYPE row_t (part_id INT, amt INT);
```

Also assume that you created a typed table named **tab1** that is based on the **row_t** ROW type:

```
CREATE TABLE tab1 OF TYPE row_t;
```

Assume also that you inserted the following values into table **tab1**:

```
INSERT INTO tab1 VALUES (ROW(1,7));
INSERT INTO tab1 VALUES (ROW(2,10));
```

Finally, assume that you created another table named **tab2**:

```
CREATE TABLE tab2 (colx INT);
```

Now you can use dot notation to insert the value from only the **part_id** column of table **tab1** into the **tab2** table:

```
INSERT INTO tab2
   VALUES ((SELECT t FROM tab1 t
      WHERE part_id = 1).part_id);
```

The asterisk form of dot notation is not necessary when you want to select all fields of a ROW-type column because you can specify the column name alone to select all of its fields. The asterisk form of dot notation can be quite helpful, however, when you use a subquery, as in the preceding example, or when you call a user-defined function to return ROW-type values.

Suppose that a user-defined function named **new_row** returns ROW-type values, and you want to call this function to insert the ROW-type values into a table. Asterisk notation makes it easy to specify that all the ROW-type values that the **new_row( )** function returns are to be inserted into the table:

```
INSERT INTO mytab2 SELECT new_row (mycol).* FROM mytab1;
```

References to the fields of a ROW-type column or a ROW-type expression are not allowed in fragment expressions. A fragment expression is an expression that defines a table fragment or an index fragment in SQL statements like CREATE TABLE, CREATE INDEX, and ALTER FRAGMENT.

Using the Substring Operator

You can use the substring operator on CHAR, VARCHAR, NCHAR, NVARCHAR, BYTE, and TEXT columns to define a *column substring* as the portion of the column that is specified by the expression.

After the identifier of a character column, when a pair of bracket ([ ]) symbols enclose a comma-separated pair of unsigned integers in which the *first* integer is greater than zero but not greater than the *last* integer, Informix® interprets the brackets as the substring operator. The expression returns the *first* through *last* characters of the data value in the column, where *first* and *last* define a substring. For example, in the expression `cat_advert [6,15]`, the returned value is the 6th through 15th characters of column **cat_advert**.

In the default locale, if the data value occupies at least 15 bytes, this expression evaluates to a substring that includes ten bytes of the column value, but in a multibyte locale this expression returns a string of ten consecutive logical characters whose storage length might exceed 10 bytes, beginning with the sixth logical character. For more information on the aspects of column substrings, see the *HCL® Informix® GLS User's Guide*.

In the following example, if a value in the **lname** column of the **customer** table is `Greenburg`, the following expression evaluates to `burg`:

```
lname[6,9]
```

A conditional expression can include a column expression that uses the substring operator ([*first*, *last*]), as in the following example:

```
SELECT lname FROM customer WHERE phone[5,7] = '356';
```

Here the quotation marks are required, to prevent the database server from applying a numeric filter to the digits in the criterion value.

See also the section String-Manipulation Functions on page 1113, which describes two built-in SQL functions, **SUBSTR( )** and **SUBSTRING( )** that can specify a substring expression within an SQL statement.

Note: The database server can use substrings defined by the substring operator as index filters in queries. This is not the case, however, for substrings defined by **SUBSTR( )** or **SUBSTRING( )**, nor for other built-in string manipulation functions.

Using Rowids

In Informix®, you can use the **rowid** column that is associated with a table row as a property of the row. The **rowid** column is essentially a hidden column in nonfragmented tables and in fragmented tables that were created with the WITH ROWIDS clause. The **rowid** column is unique for each row, but it is not necessarily sequential. It is recommended, however, that you use primary keys as an access method rather than exploiting the **rowid** column.

The following examples use the ROWID keyword in a SELECT statement:

```
SELECT *, ROWID FROM customer;

SELECT fname, ROWID FROM customer ORDER BY ROWID;

SELECT HEX(rowid) FROM customer WHERE customer_num = 106;
```

The last example shows how to get the page number (the first six digits after 0x) and the slot number (the last two digits) of the location of your row.

You cannot use the ROWID keyword in the select list of the Projection clause of a query that contains an aggregate function.

Using Smart Large Objects

The SELECT, UPDATE, and INSERT statements do not manipulate the values of smart large objects directly. Instead, they use a *handle value*, which is a type of pointer, to access the BLOB or CLOB value, as follows:

- The SELECT statement returns a handle value to the BLOB or CLOB value that the projection list specifies. SELECT does not return the actual data for the BLOB or CLOB column that the projection list specifies. Instead, it returns a handle value to the column data.
- The INSERT and UPDATE statements do not send the actual data for the BLOB or CLOB column to the database server. Instead, they accept a handle value to this data as the value to be inserted or updated.

To access the data of a smart-large-object column, you must use one of the following application programming interfaces (APIs):

- From within programs, use the library functions that access smart large objects. For more information, see the *HCL® Informix® Enterprise Replication Guide*.
- From within a C program such as a module, use the Client and Server API.

You cannot use the name of a smart-large-object column in expressions that involve arithmetic operators. For example, operations such as addition or subtraction on the smart-large-object handle value have no meaning.

When you select a smart-large-object column, you can assign the handle value to any number of columns: all columns with the same handle value share the CLOB or BLOB value. This storage arrangement reduces the amount of disk space that the CLOB or BLOB value, but when several columns share the same smart-large-object value, the following conditions result:

- The chance of lock contention on a CLOB or BLOB column increases. If two columns share the same smart-large-object value, the data might be locked by either column that needs to access it.
- The CLOB or BLOB value can be updated from a number of points.

To remove these constraints, you can create separate copies of the BLOB or CLOB data for each column that needs to access it. You can use the **LOCOPY** function to create a copy of an existing smart large object.

You can also use the built-in functions **LOTOFILE**, **FILETOCLOB**, and **FILETOBLOB** to access smart-large-object values, as described in Smart-Large-Object Functions on page 1095. For more information on the BLOB and CLOB data types, see the *HCL® Informix® Guide to SQL: Reference*.

Conditional Expressions

Conditional expressions return values that depend on the outcome of conditional tests. This diagram shows the syntax for Conditional Expressions.

(explicit id unique_1435_Connect_42_ce001) unique_1435_Connect_42_ce001(explicit id unique_1435_Connect_42_ce002)

unique_1435_Connect_42_ce002(explicit id unique_1435_Connect_42_ce005) unique_1435_Connect_42_ce005(explicit id unique_1435_Connect_42_ce004)

unique_1435_Connect_42_ce004(explicit id unique_1435_Connect_42_ce003) unique_1435_Connect_42_ce003(explicit id unique_1435_Connect_42_ce006)

unique_1435_Connect_42_ce006

Conditional Expressions

" { <CASE Expressions> | <COALESCE Function> | <NVL Function> | <NVL2 Function> | <NULLIF Function> | <DECODE Function> } "

CASE Expressions

The CASE expression allows an SQL statement such as the SELECT statement to return one of several possible results, depending on which of several conditions evaluates to true.

The CASE expression has two forms: generic CASE expressions and linear CASE expressions.

(explicit id unique_1317_Connect_42_case002a) unique_1317_Connect_42_case002a(explicit id unique_1317_Connect_42_case003)

unique_1317_Connect_42_case003

CASE Expressions

```
" { <Generic CASE Expression> [] | <Linear CASE Expression> [] } "
```

You must include at least one WHEN clause in the CASE expression. Subsequent WHEN clauses and the ELSE clause are optional. You can use a generic or linear CASE expression wherever you can use a column expression in an SQL statement (for example, in the Projection clause a SELECT statement).

Expressions in the search condition or the result value expression can contain subqueries, and you can nest a CASE expression in another CASE expression.

When a CASE expression appears in an aggregate expression, you cannot use aggregate functions in the CASE expression.

You can specify a trigger-type Boolean operator (DELETING, INSERTING, SELECTING, or UPDATING) as a condition in a CASE expression only within a trigger routine.

The following query fragment declares aliases for two aggregate column expressions:

```
SELECT . . .
   SUM(orders.ship_weight) as o2,
   COUNT(DISTINCT
         CASE WHEN orders.backlog MATCHES 'n'
              THEN orders.order_num END ) AS o3,
   . . .
```

Here the argument to **SUM** is a DECIMAL(8,2) column value, and the **COUNT DISTINCT** aggregate takes a **CASE** expression as its argument.

Do not confuse CASE expressions with the CASE statement of SPL, which supports different syntax and functionality.

CASE expressions data type compatibility

In a CASE expression, all the results should be of the same data type or be compatible data types.

If the results in all the WHEN ... THEN branch clauses are not of the same data type or compatible data types, an error occurs.

The following table shows which character data types are compatible and the data type that is returned for each combination.

Table 18. Data types returned from compatible character data types

Data type	NCHAR (>255)	NCHAR (<=255)	NVARCHAR	CHAR (<=255)	CHAR (>255)	VARCHAR	LVARCHAR (>255)	LVARCHAR (<=255)
NCHAR (>255)	NCHAR	NCHAR	NCHAR	NCHAR	NCHAR	NCHAR	NCHAR	NCHAR
NCHAR (<=255)	NCHAR	NCHAR	NVARCHAR	NCHAR	NCHAR	NVARCHAR	NCHAR	NCHAR

Table 18. Data types returned from compatible character data types (continued)

Data type	NCHAR (>255)	NCHAR (<=255)	NVARCHAR	CHAR (<=255)	CHAR (>255)	VARCHAR	LVARCHAR (>255)	LVARCHAR (<=255)
NVARCHAR	NCHAR	NVARCHAR	NVARCHAR	NVARCHAR	NCHAR	NVARCHAR	NCHAR	NVARCHAR
CHAR (<=255)	NCHAR	NCHAR	NVARCHAR	CHAR	CHAR	VARCHAR	CHAR	CHAR
CHAR (>255)	NCHAR	NCHAR	NCHAR	CHAR	CHAR	CHAR	CHAR	CHAR
VARCHAR	NCHAR	NVARCHAR	NVARCHAR	VARCHAR	CHAR	VARCHAR	CHAR	VARCHAR
LVARCHAR (>255)	NCHAR	NCHAR	NCHAR	CHAR	CHAR	CHAR	LVARCHAR	LVARCHAR
LVARCHAR (<=255)	NCHAR	NCHAR	NVARCHAR	CHAR	CHAR	VARCHAR	LVARCHAR	LVARCHAR

The following table shows which numeric data types are compatible and the data type that is returned for each combination.

Table 19. Data types returned from compatible numeric data types

Data type	INTEGER	SMALLINT	SERIAL	DECIMAL	FLOAT	SMALLFLOAT	MONEY	BIGINT	BIGSERIAL
INTEGER	INTEGER	INTEGER	INTEGER	DECIMAL	DECIMAL	DECIMAL	MONEY	DECIMAL	DECIMAL
SMALLINT	INTEGER	SMALLINT	INTEGER	DECIMAL	DECIMAL	DECIMAL	MONEY	DECIMAL	DECIMAL
SERIAL	INTEGER	INTEGER	SERIAL	DECIMAL	DECIMAL	DECIMAL	MONEY	DECIMAL	DECIMAL
DECIMAL	DECIMAL	DECIMAL	DECIMAL	DECIMAL	DECIMAL	DECIMAL	MONEY	DECIMAL	DECIMAL
FLOAT	DECIMAL	DECIMAL	DECIMAL	DECIMAL	FLOAT	FLOAT	MONEY	DECIMAL	DECIMAL
SMALLFLOAT	DECIMAL	DECIMAL	DECIMAL	DECIMAL	FLOAT	SMALLFLOAT	MONEY	DECIMAL	DECIMAL
MONEY	MONEY	MONEY	MONEY	MONEY	MONEY	MONEY	MONEY	MONEY	MONEY
BIGINT	DECIMAL	DECIMAL	DECIMAL	DECIMAL	DECIMAL	DECIMAL	MONEY	BIGINT	BIGINT
BIGSERIAL	DECIMAL	DECIMAL	DECIMAL	DECIMAL	DECIMAL	DECIMAL	MONEY	BIGINT	BIGSERIAL

Generic CASE Expressions

A generic CASE expression tests for a true condition in a WHEN clause. If it finds a true condition, it returns the result specified in the THEN clause.

(explicit id unique_1611_Connect_42_gce002) unique_1611_Connect_42_gce002

Generic CASE Expression

" **CASE** "

" **WHEN** <Condition>[] **THEN** { *expr* | **NULL** } "

" [**ELSE** { *expr* | **NULL** }] "

" **END** "

Element	Description	Restrictions	Syntax
expr	Expression that returns some data type	Data type of *expr* in a THEN clause must be compatible with data types of expressions in other THEN clauses	Expression on page 1003

The database server processes the WHEN clauses in the order that they appear in the statement. If the search condition of a WHEN clause evaluates to `TRUE`, the database server uses the value of the corresponding THEN expression as the result, and stops processing the CASE expression.

If no WHEN condition evaluates to `TRUE`, the database server uses the ELSE expression as the overall result. If no WHEN condition evaluates to `TRUE`, and no ELSE clause was specified, the returned CASE expression value is NULL. You can use the IS NULL condition to handle NULL results. For information on how to handle NULL values, see IS NULL and IS NOT NULL Conditions on page 974.

The next example shows a generic CASE expression in the Projection clause.

In this example, the user retrieves the name and address of each customer as well as a calculated number that is based on the number of problems that exist for that customer:

```
SELECT cust_name,
   CASE
   WHEN number_of_problems = 0
      THEN 100
   WHEN number_of_problems > 0 AND number_of_problems < 4
      THEN number_of_problems * 500
   WHEN number_of_problems >= 4 and number_of_problems <= 9
      THEN number_of_problems * 400
   ELSE
      (number_of_problems * 300) + 250
   END,
   cust_address
FROM custtab
```

In a generic CASE expression, all the results should be of the same data type, or they should evaluate to a common compatible data type. If the results in all the WHEN clauses are not of the same data type, or if they do not evaluate to values of mutually compatible types, an error occurs. For more information on the compatibility of returned data types, see CASE expressions data type compatibility on page 1033.

Linear CASE Expressions

A linear CASE expression compares the value of the expression that follows the CASE keyword with an expression in a WHEN clause.

Linear CASE Expression

" CASE *expr* "

" WHEN *expr* THEN { *expr* | NULL } "

" [ELSE { *expr* | NULL }] "

" END "

Element	Description	Restrictions	Syntax
expr	Expression that returns a value of some data type	Data type of *expr* that follows the WHEN keyword must be compatible with data type of the expression that follows the CASE keyword. Data type of *expr* in the THEN clause must be compatible with data types of expressions in other THEN clauses.	Expression on page 1003

The database server evaluates the expression that follows the CASE keyword, and then processes the WHEN clauses sequentially. If an expression after the WHEN keyword returns the same value as the expression that follows the CASE keyword, the database server uses the value of the expression that follows the THEN keyword as the overall result of the CASE expression. Then the database server stops processing the CASE expression.

If none of the WHEN expressions return the same value as the expression that follows the CASE keyword, the database server uses the expression of the ELSE clause as the overall result of the CASE expression (or, if no ELSE clause was specified, the returned value of the CASE expression is NULL).

The next example shows a linear CASE expression in the projection list of the Projection clause of a SELECT statement. For each movie in a table of movie titles, the query returns the title, the cost, and the type of the movie. The statement uses a CASE expression to derive the type of each movie:

```
SELECT title, CASE movie_type
      WHEN 1 THEN 'HORROR'
      WHEN 2 THEN 'COMEDY'
      WHEN 3 THEN 'ROMANCE'
      WHEN 4 THEN 'WESTERN'
      ELSE 'UNCLASSIFIED'
   END,
   our_cost FROM movie_titles;
```

In linear CASE expressions, the data types of WHEN clause expressions must be compatible with that of the expression that follows the CASE keyword.

COALESCE Function

The COALESCE function returns the first non-NULL value from a series of expressions.

COALESCE Function

" COALESCE (*expression* [, *expression*]) "

Element	Description	Restrictions	Syntax
expression	An expression that returns a value of a compatible data type with the other expressions in the function	Cannot be a host variable or a BYTE or TEXT object	Expression on page 1003

The expressions are evaluated in the order in which they are specified, and the result of the function is the first value that is not null. The result of the COALESCE function returns NULL only if all the arguments are null. The expressions can return any data type that can be cast to a common compatible data type.

Suppose that the **addr** column of the **employees** table has NULL values in some rows, and that you want to print the label `Address unknown` for these rows. Run the following SELECT statement to display the label `Address unknown` when the **addr** column has a NULL value:

```
SELECT fname, COALESCE (addr, 'Address unknown') AS address
   FROM employees;
```

NVL Function

The **NVL** expression returns different results, depending on whether its first argument evaluates to NULL.

NVL Function

" NVL(*expr1, expr2*) "

Element	Description	Restrictions	Syntax
expr1 *expr2*	Expressions that return values of a compatible data type	Cannot be a host variable or a BYTE or TEXT object	Expression on page 1003

NVL evaluates *expression1*. If *expression1* is not NULL, then **NVL** returns the value of *expression1*. If *expression1* is NULL, **NVL** returns the value of *expression2*. The expressions *expression1* and *expression2* can be of any data type, as long as they can be cast to a common compatible data type.

Suppose that the **addr** column of the **employees** table has NULL values in some rows, and the user wants to be able to print the label `Address unknown` for these rows. The user enters the following SELECT statement to display the label `Address unknown` when the **addr** column has a NULL value:

```
SELECT fname, NVL (addr, 'Address unknown') AS address
   FROM employees;
```

NULLIF Function

The **NULLIF** expression returns different results, depending on whether its two arguments are equal.

NULLIF Function

" **NULLIF**(*expr1*, *expr2*) "

Element	Description	Restrictions	Syntax
expr1 *expr2*	Expressions that return values of a compatible data type	Cannot be a BYTE or TEXT data type	Expression on page 1003

NULLIF evaluates its two arguments, *expr1* and *expr2*.

- If their values are equal. then **NULLIF** returns NULL.
- If their values are not equal. then **NULLIF** returns *expr1*.

The *expr1* and *expr2* arguments can be of any data type for which a built-in comparison function exists, or any two data types that can be cast to a compatible data type that has a built-in comparison function.

The following example uses the **NULLIF** function to convert Boolean FALSE values ('f') to NULL values:

```
SELECT name, answer, NULLIF(answer, 'f') FROM booktab;
```

Here the first argument is a Boolean column expression that can have true ('t') or false ('f') values, and the second Boolean argument is always 'f" (for FALSE). For rows that have 'f' in the **answer** column, the value returned by the **NULLIF** function will be NULL (because the NULL value is returned when the arguments are equal). For rows that have 't' as the first argument, however, the value returned by **NULLIF** is always 't', because the two arguments cannot be equal when one is 't' and the other is 'f'; the first argument is returned when the two values are not equal.

DECODE Function

The **DECODE** expression is similar to the CASE expression in that it can print different results depending on the values found in a specified column.

DECODE Function

" **DECODE**(*expr*, *when_expr*, { *then_expr* | **NULL** } [{ , **NULL** | [, *else_expr*] }]) "

Element	Description	Restrictions	Syntax
expr, *else_expr*, *then_expr*, *when_expr*	Expressions whose values and data types can be evaluated	Data types of *when_expr* and *expr* must be compatible, as must *then_expr* and *else_expr*. Value of *when_expr* cannot be a NULL.	Expression on page 1003

The expressions *expr*, *when_expr*, and *then_expr* are required. **DECODE** evaluates *expr* and compares it to *when_expr*. If the value of *when_expr* matches the value of *expr*, then **DECODE** returns *then_expr*.

The expressions *when_expr* and *then_expr* are an expression pair, and you can specify any number of expression pairs in the **DECODE** function. In all cases, **DECODE** compares the first member of the pair against *expr* and returns the second member of the pair if the first member matches *expr*.

If no expression matches *expr*, **DECODE** returns *else_expr*. If no expression matches *expr* and you specified no *else_expr*, then **DECODE** returns NULL.

You can specify any data type for the arguments, but two restrictions exist:

- All instances of *when_expr* must have the same data type, or a common compatible type must exist. All instances of *when_expr* must also have the same (or a compatible) data type as *expr*.
- All instances of *then_expr* must have the same data type, or a common compatible type must exist. All instances of *then_expr* must also have the same (or a compatible) data type as *else_expr*.

The DECODE function uses the same data type compatibility rules as a CASE expression. For more information on the compatibility of returned data types, see CASE expressions data type compatibility on page 1033.

Example

Suppose that a user wants to convert descriptive values in the **evaluation** column of the **students** table to numeric values in the output. The following table shows the contents of the **students** table.

firstname	evaluation	firstname	evaluation
Edward	Great	Mary	Good
Joe	Not done	Jim	Poor

The user now enters a query with the **DECODE** function to convert the descriptive values in the **evaluation** column to numeric equivalents:

```
SELECT firstname, DECODE(evaluation,
   'Poor', 0,
   'Fair', 25,
   'Good', 50,
   'Very Good', 75,
   'Great', 100,
   -1) as grade
FROM students;
```

The following table shows the output of this SELECT statement.

firstname	evaluation	firstname	evaluation
Edward	100	Mary	50
Joe	-1	Jim	0

Constant Expressions

Certain expressions that return a fixed value are called *constant expressions*. These include variant function operators that read the system clock, but that are valid in contexts where literal constants are also valid.

Among these expressions are the following operators (or *system constants*) whose returned values are determined at runtime:

- **CURRENT** returns the current time and date from the system clock.
- **CURRENT_ROLE** returns the name of the role, if any, whose privileges are enabled for the current user.
- **CURRENT_USER** is a synonym for **USER**.
- **DEFAULT_ROLE** returns the name of the role, if any, that is the default role for the current user.
- **DBSERVERNAME** returns the name of the current database server.
- **SITENAME** is a synonym for **DBSERVERNAME.**
- **SYSDATE** reads the DATETIME value from the system clock like the **CURRENT** operator, but has a different default precision.
- **TODAY** returns the current calendar date from the system clock.
- **USER** returns the login name (also called the *authorization identifier*) of the current user.

Besides these operators, the term *constant expression* can also refer to a quoted string, to a literal value, or to the **UNITS** operator with its operands.

The Constant Expression segment has the following syntax.

(explicit id unique_60_Connect_42_ref1) unique_60_Connect_42_ref1(explicit id unique_60_Connect_42_ref2) unique_60_Connect_42_ref2(explicit id unique_60_Connect_42_ref3) unique_60_Connect_42_ref3(explicit id unique_60_Connect_42_ref5) unique_60_Connect_42_ref5(explicit id unique_60_Connect_42_ref6) unique_60_Connect_42_ref6(explicit id unique_60_Connect_42_ref8) unique_60_Connect_42_ref8(explicit id unique_60_Connect_42_ref10) unique_60_Connect_42_ref10(explicit id unique_60_Connect_42_ref9) unique_60_Connect_42_ref9

Constant Expressions

```
" { <Quoted String> [] | <Literal Number> [] | { { USER | CURRENT_USER } | [] { CURRENT_ROLE | DEFAULT_ROLE } } | { []
| { SITENAME | DBSERVERNAME } | TODAY | { CURRENT | SYSDATE } [ precision ] | <Literal DATETIME> [] | <Literal
INTERVAL> [] | num UNITStime_unit | { [ owner . ] { sequence | synonym } . { CURRVAL | NEXTVAL } | <Literal
Collection> [] | <Literal Row> [] | literal opaque type | literal BOOLEAN } } } "
```

Element	Description	Restrictions	Syntax
literal Boolean	Literal representation of a BOOLEAN value	Must be either *t* (TRUE) or *f* (FALSE)	Quoted String on page 1202
literal opaque type	Literal representation of value of an opaque data type	Must be recognized by the input support function of opaque type	Defined by UDT developer
num	How many of specified time units. See UNITS Operator on page 1046.	If *num* is not an integer, the fractional part is truncated	Literal Number on page 1198
owner	Name of the owner of sequence	Must own sequence	Owner name on page 1262
precision	Precision of the returned DATETIME expression	On Windows™ systems the maximum scale of seconds is FRACTION(3).	#unique_162
sequence	Name of a sequence	Must exist in current database	Identifier on page 1231
synonym	Synonym for the name of a sequence	Must exist in current database	Identifier on page 1231
time_unit	Keyword to specify time unit: YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, *or* FRACTION	Must be one of the keywords at left. Case insensitive but cannot be enclosed within quotes	See the **Restrictions** column.

Quoted String

The following examples show quoted strings as expressions:

```
SELECT 'The first name is ', fname FROM customer;

INSERT INTO manufact VALUES ('SPS', 'SuperSport');

UPDATE cust_calls SET res_dtime = '2007-1-1 10:45'
   WHERE customer_num = 120 AND call_code = 'B';
```

For more information, see Quoted String on page 1202.

Literal Number

A literal number specifies a numeric value.

The following examples show literal numbers as expressions:

```
INSERT INTO items VALUES (4, 35, 52, 'HRO', 12, 4.00);

INSERT INTO acreage VALUES (4, 5.2e4);

SELECT unit_price + 5 FROM stock;
```

```
SELECT -1 * balance FROM accounts;
```

For more information, see Literal Number on page 1198.

USER or CURRENT_USER Operator

The **USER** operator returns a string containing the login name (also called the authorization identifier) of the current user who is running the process. The **CURRENT_USER** operator is a synonym of the **USER** operator.

The following statements show how you might use the **USER** operator:

```
INSERT INTO cust_calls VALUES
   (221,CURRENT,USER,'B','Decimal point off', NULL, NULL);

SELECT * FROM cust_calls WHERE user_id = USER;

UPDATE cust_calls SET user_id = USER WHERE customer_num = 220;
```

The **USER** operator does not change the lettercase of a user ID. If you use **USER** in an expression and the current user is **Robertm**, the **USER** operator returns **Robertm**, not **robertm** or **ROBERTM**.

If you specify **USER** as a default column value, *column* must be of type CHAR, VARCHAR, NCHAR, NVARCHAR, or LVARCHAR.

If you specify **USER** as the default value for a column, the size of *column* should not be less than 32 bytes. You risk getting an error during operations such as INSERT or ALTER TABLE if the column length is too small to store the default value.

In an ANSI-compliant database, if you do not enclose the *owner* name in quotation marks, the name of the table owner is stored as uppercase letters. If you use the **USER** operator as part of a condition, you must be sure that the way the *user* name is stored matches what the **USER** operator returns with respect to lettercase.

CURRENT_ROLE Operator

The **CURRENT_ROLE** operator returns a string that contains the name of the currently enabled role of the user who is running the session. This *role* was either set in the session explicitly, using the SET ROLE statement, or else implicitly as a default role when the current user connected to the database. If the user holds no role, or if no role that was granted to the user is currently enabled, **CURRENT_ROLE** returns a NULL value. If the user has been granted no role individually, but a default role has been granted to PUBLIC, and this default role has been explicitly or implicitly enabled, **CURRENT_ROLE** returns the name of this default role.

The next statement shows how you might use the **CURRENT_ROLE** operator:

```
select CURRENT_ROLE FROM systables WHERE tabid = 1;
```

The **CURRENT_ROLE** operator does not change the lettercase of the identifier of a role. If you use **CURRENT_ROLE** in an expression and your current role is **Czarina**, the **CURRENT_ROLE** operator returns **Czarina**, not **czarina**.

If you specify **CURRENT_ROLE** as the default value for a column, the column must have a CHAR, VARCHAR, LVARCHAR, NCHAR, or NVARCHAR data type. Because the name of a role is an authorization identifier, truncation might occur if the column length is less than 32 bytes.

DEFAULT_ROLE Operator

The **DEFAULT_ROLE** operator evaluates to a string that contains the name of the default role that has been granted to the user who is running the session. This default role need not be currently enabled, but it must not have been revoked since the most recent GRANT DEFAULT ROLE statement that referenced the user or PUBLIC in the TO clause.

If no default role is explicitly defined for the current user, but PUBLIC has a default role, **DEFAULT_ROLE** returns the default role of PUBLIC.

If the user has no default role, or if the default role that was most recently granted to the user explicitly, or as PUBLIC, was subsequently revoked by the REVOKE DEFAULT ROLE statement, **DEFAULT_ROLE** returns a NULL value. If the user has been granted no default role individually, but a default role has been granted to PUBLIC, the **DEFAULT_ROLE** operator returns the name of this default role. If no default role is currently defined for the user nor for PUBLIC, however, **DEFAULT_ROLE** returns NULL.

The SET ROLE statement has no effect on the **DEFAULT_ROLE** operator, but any access privileges of the default role are not necessarily available to the user if SET ROLE has activated some other role, or if SET ROLE specified NULL or NONE as the current role of the user.

The next statements show how you might use the **DEFAULT_ROLE** operator:

```
select DEFAULT_ROLE from systables where tabid = 1;
```

DEFAULT_ROLE does not change the lettercase of the identifier of a role.

If you specify **DEFAULT_ROLE** as the default value for a column, the column must have a CHAR, VARCHAR, LVARCHAR, NCHAR, or NVARCHAR data type. Because the name of a role is an authorization identifier, truncation might occur if the column width is less than 32 bytes. (See Owner name on page 1262 for the syntax of authorization identifiers.)

DBSERVERNAME and SITENAME Operators

The **DBSERVERNAME** operator returns the SQL identifier of the database server, as defined by the DBSERVERNAME parameter in the ONCONFIG file for the Informix® instance where the current database resides, or as specified in the **INFORMIXSERVER** environment variable. **SITENAME** is a keyword synonym for the **DBSERVERNAME** operator.

You can use the **DBSERVERNAME** operator to specify the location of a table, to put information into a table, or to extract information from a table. You can insert **DBSERVERNAME** into a simple character field or use it as a default value for a column.

If you specify **DBSERVERNAME** as a default column value in the CREATE TABLE or ALTER TABLE statements, the column must be a CHAR, VARCHAR, LVARCHAR, NCHAR, or NVARCHAR data type.

If you specify **DBSERVERNAME** or **SITENAME** as the default value for a column, the size of the column should be at least 128 bytes long. You risk getting an error message during INSERT and ALTER TABLE operations if the length of the column is too small to store the default value.

The following examples use **DBSERVERNAME** or **SITENAME** in DML statements.

- The first SELECT statement returns the name of the database server instance where the **customer** table resides. (Because the query is not restricted by a WHERE clause, it returns the same **DBSERVERNAME** value for every row in the table. If you include the DISTINCT keyword in the projection clause, the query returns **DBSERVERNAME** only once.)
- The second statement adds a row that contains the name of the current database server to a table.
- The third statement returns all rows that have the name of the current database server in the **host_tab.site_col** column.
- The last statement changes to the name of the current database server the value of the **customer.company** column in the row whose SERIAL value of **customer_num** is `120`:

```
SELECT DBSERVERNAME FROM customer;

INSERT INTO host_tab VALUES ('1', SITENAME);

SELECT * FROM host_tab WHERE site_col = DBSERVERNAME;

UPDATE customer SET company = SITENAME
    WHERE customer_num = 120;
```

TODAY Operator

Use the **TODAY** operator to return the system date as a DATE data type. If you specify **TODAY** as a default column value, the column must be a DATE column.

The following examples show how you might use the **TODAY** operator in an INSERT, UPDATE, or SELECT statement:

```
UPDATE orders (order_date) SET order_date = TODAY
    WHERE order_num = 1005;

INSERT INTO orders VALUES
   (0, TODAY, 120, NULL, N, '1AUE217', NULL, NULL, NULL, NULL);

SELECT * FROM orders WHERE ship_date = TODAY;
```

For code examples of setting non-default time zones, see CURRENT Operator on page 1044.

CURRENT Operator

The **CURRENT** operator returns a DATETIME value with the date and time of day, showing the current instant.

If you do not specify a DATETIME qualifier, the default qualifier is YEAR TO FRACTION(3). The USEOSTIME configuration parameter specifies whether or not the database server uses subsecond precision when it obtains the current time from the operating system. For more information on the USEOSTIME configuration parameter, see your *HCL® Informix® Administrator's Reference*.

You can use **CURRENT** in any context where a literal DATETIME is valid. (See #unique_474). If you specify **CURRENT** as the default value for a column, it must be a DATETIME column and the qualifier of **CURRENT** must match the column qualifier, as the following example shows:

```
CREATE TABLE new_acct (col1 INT, col2 DATETIME YEAR TO DAY
   DEFAULT CURRENT YEAR TO DAY);
```

CURRENT is always evaluated in the database server where the current database is located. If the current database is in a remote database server, the returned value is from the remote host.

SQL is not a procedural language, and **CURRENT** might not execute in the lexical order of its position in a statement. You should not use **CURRENT** to mark the start, the end, nor a specific point in the execution of an SQL statement.

If you use the **CURRENT** operator in more than once in a single statement, identical values might be returned by each instance of CURRENT. You cannot rely on **CURRENT** to return distinct values each time it executes.

The returned value is based on the system clock and is fixed when the SQL statement that specifies CURRENT starts execution. For example, any call to **CURRENT** from inside the SPL function that an EXECUTE FUNCTION (or EXECUTE PROCEDURE) statement invokes returns the value of the system clock when the SPL function starts.

On UNIX™ and Linux™ systems, the precision of the value returned by the CURRENT operator is determined by its DATETIME Qualifier, which can range from a single time unit (such as MONTH TO MONTH) up to YEAR TO FRACTION (5). The system clock on Windows™, however, returns only millisecond precision. Even if you specify "FRACTION(5)" in the DATETIME Qualifier, the CURRENT operator on Windows™ supports no greater than "FRACTION(3)" precision.

If your platform does not provide a system call that returns the current time with subsecond precision, **CURRENT** returns a zero for the FRACTION field.

In the following example, the first statement uses **CURRENT** in a WHERE condition. The second statement uses **CURRENT** as an argument to the **DAY** function. The last query selects rows whose **call_dtime** value is within a range from the beginning of 2007 to the current instant:

```
DELETE FROM cust_calls WHERE res_dtime < CURRENT YEAR TO MINUTE;

SELECT * FROM orders WHERE DAY(ord_date) < DAY(CURRENT);

SELECT * FROM cust_calls WHERE call_dtime
   BETWEEN '2007-1-1 00:00:00' AND CURRENT;
```

For more information, see #unique_162.

Literal DATETIME

A literal DATETIME specifies the value of an DATETIME data type, including its qualifying time-units.

The following examples show literal DATETIME values as expressions:

```
SELECT DATETIME (2007-12-6) YEAR TO DAY FROM customer;

UPDATE cust_calls SET res_dtime = DATETIME (2008-07-07 10:40)
         YEAR TO MINUTE
   WHERE customer_num = 110
   AND call_dtime = DATETIME (2008-07-07 10:24) YEAR TO MINUTE;

SELECT * FROM cust_calls
```

```
   WHERE call_dtime
   = DATETIME (2008-12-25 00:00:00) YEAR TO SECOND;
```

For more information, see #unique_474.

Literal INTERVAL

A literal INTERVAL specifies the value of an INTERVAL data type, including its qualifying time-units.

The following examples each use a literal INTERVAL as an expression:

```
INSERT INTO manufact VALUES ('CAT', 'Catwalk Sports',
   INTERVAL (16) DAY TO DAY);

SELECT lead_time + INTERVAL (5) DAY TO DAY FROM manufact;
```

The second example adds five days to each value of **lead_time** selected from the **manufact** table.

For more information, see #unique_475.

UNITS Operator

The UNITS operator specifies an INTERVAL value whose precision includes only one time unit. You can use UNITS in arithmetic expressions that increase or decrease one of the time units in an INTERVAL or DATETIME value.

If the *num* operand is not an integer, it is truncated to the largest whole number that is the same as (or nearer to zero than) the specified value when the database server evaluates the expression.

In the following example, the first SELECT statement uses the UNITS operator to select all the **manufacturer.lead_time** values, increased by five days. The second SELECT statement finds all the calls that were placed more than 30 days ago.

If the expression in the WHERE clause returns a value greater than 99 (maximum number of days), the query fails. The last statement increases the lead time for the ANZA manufacturer by two days:

```
SELECT lead_time + 5 UNITS DAY FROM manufact;

SELECT * FROM cust_calls WHERE (TODAY - call_dtime) > 30 UNITS DAY;

UPDATE manufact SET lead_time = 2 UNITS DAY + lead_time
   WHERE manu_code = 'ANZ';
```

NEXTVAL and CURRVAL Operators

You can access the value of a sequence using the **NEXTVAL** or **CURRVAL** operators in SQL statements. You must qualify **NEXTVAL** or **CURRVAL** with the name (or synonym) of a sequence object that exists in the same database, using the format *sequence*.**NEXTVAL** or *sequence*.**CURRVAL**. An expression can also qualify *sequence* by the *owner* name, as in **zelaine.myseq.CURRVAL**. You can specify the SQL identifier of *sequence* or a valid synonym, if one exists.

In an ANSI-compliant database, you must qualify the name of the *sequence* with the name of its owner (*owner.sequence*) if you are not the owner.

To use **NEXTVAL** or **CURRVAL** with a sequence, you must have the Select privilege on the sequence or have the DBA privilege on the database. For information about sequence-level privileges, see the GRANT statement on page 537 statement.

Examples

In the following examples, it is assumed that no other user is concurrently accessing the sequence and that the user executes the statements consecutively.

These examples are based on the following sequence object and table:

```
CREATE SEQUENCE seq_2
   INCREMENT BY 1 START WITH 1
   MAXVALUE 30 MINVALUE 0
   NOCYCLE CACHE 10 ORDER;

CREATE TABLE tab1 (col1 int, col2 int);
INSERT INTO tab1 VALUES (0, 0);
```

You can use **NEXTVAL** (or **CURRVAL**) in the Values clause of an INSERT statement, as the following example shows:

```
INSERT INTO tab1 (col1, col2)
   VALUES (seq_2.NEXTVAL, seq_2.NEXTVAL);
```

In the previous example, the database server inserts an incremented value (or the first value of the sequence, which is `1`) into the **col1** and **col2** columns of the table.

You can use **NEXTVAL** (or **CURRVAL**) in the SET clause of the UPDATE statement, as the following example shows:

```
UPDATE tab1
   SET col2 = seq_2.NEXTVAL
   WHERE col1 = 1;
```

In the previous example, the incremented value of the **seq_2** sequence, which is `2`, replaces the value in **col2** where **col1** is equal to `1`.

The following example shows how you can use **NEXTVAL** and **CURRVAL** in the Projection clause of the SELECT statement:

```
SELECT seq_2.CURRVAL, seq_2.NEXTVAL FROM tab1;
```

In the previous example, the database server returns two rows of incremented values, `3` and `4`, from both the **CURRVAL** and **NEXTVAL** expressions. For the first row of **tab1**, the database server returns the incremented value `3` for **CURRVAL** and **NEXTVAL**; for the second row of **tab1**, it returns the incremented value `4`.

Using NEXTVAL

To access a sequence for the first time, you must refer to *sequence*.**NEXTVAL** before you can refer to *sequence.***CURRVAL**. The first reference to **NEXTVAL** returns the initial value of the sequence. Each subsequent reference to **NEXTVAL** increments the value of the sequence by the defined *step* and returns a new incremented value of the sequence.

You can increment a given sequence only once within a single SQL statement. Even if you specify *sequence***.NEXTVAL** more than once within a single statement, the sequence is incremented only once, so that every occurrence of *sequence***.NEXTVAL** in the same SQL statement returns the same value.

Except for the case of multiple occurrences within the same statement, every *sequence*.**NEXTVAL** expression increments the *sequence*, regardless of whether you subsequently commit or roll back the current transaction.

If you specify *sequence*.**NEXTVAL** in a transaction that is ultimately rolled back, some sequence numbers might be skipped.

Using CURRVAL

Any reference to **CURRVAL** returns the current value of the specified sequence, which is the value that your last reference to **NEXTVAL** returned. After you generate a new value with **NEXTVAL**, you can continue to access that value using **CURRVAL**, regardless of whether another user increments the sequence.

If both *sequence*.**CURRVAL** and *sequence*.**NEXTVAL** occur in an SQL statement, the sequence is incremented only once. In this case, each *sequence*.**CURRVAL** and *sequence*.**NEXTVAL** expression returns the same value, regardless of the order of *sequence*.**CURRVAL** and *sequence*.**NEXTVAL** within the statement.

Concurrent Access to a Sequence

A sequence always generates unique values within a database without perceptible waiting or locking, even when multiple users refer to the same sequence concurrently. When multiple users use **NEXTVAL** to increment the sequence, each user generates a unique value that other users cannot see.

When multiple users concurrently increment the same sequence, gaps occur between the values that each user sees. For example, one user might generate a series of values, such as `1`, `4`, `6`, and `8`, from a sequence, while another user concurrently generates the values `2`, `3`, `5`, and `7` from the same sequence object.

Restrictions on sequence operators

NEXTVAL and **CURRVAL** are valid only in SQL statements, not directly in SPL statements. (But SQL statements that use **NEXTVAL** and **CURRVAL** can be used in SPL routines.) The following restrictions apply to these operators in SQL statements:

- You must have Select privilege on the *sequence*.
- In a CREATE TABLE or ALTER TABLE statement, you cannot specify **NEXTVAL** or **CURRVAL** in the following contexts:
    - In the Default clause of a column definition
    - In the definition of a check constraint.
- In a SELECT statement, you cannot specify **NEXTVAL** or **CURRVAL** in the following contexts:
    - In the projection list when the DISTINCT keyword is used
    - In the WHERE, GROUP BY, or ORDER BY clauses
    - In a subquery
    - When the UNION operator combines SELECT statements.
- You also cannot specify **NEXTVAL** or **CURRVAL** in these contexts:
    - In fragmentation expressions
    - In reference to a remote sequence object in another database.

Literal Row

The syntax for a literal representation of the value of a named or unnamed ROW data type is described in the section Literal Row on page 1199. The following examples show literal rows as expressions:

```
INSERT INTO employee VALUES
   (ROW('103 Baker St', 'San Francisco',
      'CA', 94500));

UPDATE rectangles
   SET rect = ROW(8, 3, 7, 20)
   WHERE area = 140;

EXEC SQL update table(:a_row)
   set x=0, y=0, length=10, width=20;

SELECT row_col FROM tab_b
   WHERE ROW(17, 'abc') IN (row_col);
```

For the syntax of expressions that evaluate to field values of a ROW data type, see ROW constructors on page 1050.

Literal Collection

Informix® supports expressions that are literal representations of the values of built-in or user-defined collection data types. The following examples show literal collections as expressions:

```
INSERT INTO tab_a (set_col) VALUES ("SET{6, 9, 3, 12, 4}");

INSERT INTO TABLE(a_set) VALUES (9765);

UPDATE table1 SET set_col = "LIST{3}";

SELECT set_col FROM table1
   WHERE SET{17} IN (set_col);
```

For more information, see Literal Collection on page 1195. For the syntax of element values, see Collection Constructors on page 1051.

Constructor Expressions

A *constructor* is a function that the database server uses to create an instance of a specific data type. The database server supports ROW constructors and collection constructors.

(explicit id unique_835_Connect_42_consex001) unique_835_Connect_42_consex001(explicit id unique_835_Connect_42_consex002)

unique_835_Connect_42_consex002

Constructor Expressions

" { ROW(<Expression> []) | <Collection Constructors> [] } "

ROW constructors

You use ROW constructors to generate values for ROW-type columns.

Suppose you create the following named ROW type and a table that contains the named ROW type **row_t** and an unnamed ROW type:

```
CREATE ROW TYPE row_t ( x INT, y INT);
CREATE TABLE new_tab
(
col1 row_t,
col2 ROW( a CHAR(2), b INT)
);
```

When you define a column as a named ROW type or unnamed ROW type, you must use a ROW constructor to generate values for the ROW-type column. To create a value for either a named ROW type or unnamed ROW type, you must complete the following steps:

- Begin the expression with the ROW keyword.
- Specify a value for each field of the ROW type.
- Enclose the comma-separated list of field values within parentheses.

The format of the value for each field must be compatible with the data type of the ROW field to which it is assigned.

You can use any kind of expression as a value with a ROW constructor, including literals, functions, and variables. The following examples show the use of different types of expressions with ROW constructors to specify values:

```
ROW(5, 6.77, 'HMO')

ROW(col1.lname, 45000)

ROW('john davis', TODAY)

ROW(USER, SITENAME)
```

The following statement uses literal numbers and quoted strings with ROW constructors to insert values into **col1** and **col2** of the **new_tab** table:

```
INSERT INTO new_tab
VALUES
(
ROW(32, 65)::row_t,
ROW('CA', 34)
);
```

When you use a ROW constructor to generate values for a named ROW type, you must explicitly cast the ROW value to the appropriate named ROW type. The cast is necessary to generate a value of the named ROW type. To cast the ROW value as a named ROW type, you can use the cast operator (**::**) or the CAST AS keywords, as the following examples show:

```
ROW(4,5)::row_t
CAST (ROW(3,4) AS row_t)
```

You can use a ROW constructor to generate ROW type values in INSERT, UPDATE, and SELECT statements. In the next example, the WHERE clause of a SELECT statement specifies a ROW type value that is cast as type **person_t**:

```
SELECT * FROM person_tab
   WHERE col1 = ROW('charlie','hunter')::person_t;
```

For more information on using ROW constructors in INSERT and UPDATE statements, see the INSERT and UPDATE statements in this document. For information on named ROW types, see the CREATE ROW TYPE statement. For information on unnamed ROW types, see the discussion of the ROW data type in the *HCL® Informix® Guide to SQL: Reference*. For task-oriented information on named ROW types and unnamed ROW types, see the *IBM® Informix® Database Design and Implementation Guide*.

Collection Constructors

Use a collection constructor to specify values for a collection column.

(explicit id unique_833_Connect_42_collco002) unique_833_Connect_42_collco002

Collection Constructors

"{ SET | MULTISET | LIST }"

"{ [<Expression>] }"

You can use collection constructors in the WHERE clause of the SELECT statement and the VALUES clause of the INSERT statement. You can also pass collection constructors to UDRs.

This table differentiates the types of collections that you can construct.

Keyword	Description
SET	Indicates a collection of elements with the following qualities: • The collection must contain unique values. • Elements have no specific order associated with them.
MULTISET	Indicates a collection of elements with the following qualities: • The collection can contain duplicate values. • Elements have no specific order associated with them.
LIST	Indicates a collection of elements with the following qualities: • The collection can contain duplicate values. • Elements have ordered positions.

The element type of the collection can be any built-in or extended data type. You can use any kind of expression with a collection constructor, including literals, functions, and variables.

When you use a collection constructor with a list of expressions, the database server evaluates each expression to its equivalent literal form and uses the literal values to construct the collection.

You specify an empty collection with a set of empty braces (**{}**).

Elements of a collection cannot be NULL. If a collection element evaluates to a NULL value, the database server returns an error.

The element type of each expression must all be exactly the same data type. To accomplish this, cast the entire collection constructor expression to a collection type, or cast individual element expressions to the same type. If the database server cannot determine that the collection type and the element types are homogeneous, then the collection constructor returns an error. In the case of host variables, this determination is made at bind time when the client declares the element type of the host variable.

An exception to this restriction can occur when some elements of a collection are VARCHAR data types but others are longer than 255 bytes. Here the collection constructor can assign a CHAR(*n*) type to all elements, for *n* the length in bytes of the longest element. (But see Collection Data Types on page 1001 for an example based on this exception, where the user avoids fixed-length CHAR elements by an explicit cast to the LVARCHAR data type.)

Examples of Collection Constructors

The following example shows that you can construct a collection with various expressions, if the resulting values are of the same data type:

```
CREATE FUNCTION f (a int) RETURNS int;
   RETURN a+1;
END FUNCTION;
CREATE TABLE tab1 (x SET(INT NOT NULL));
INSERT INTO tab1 VALUES
(
SET{10,
   1+2+3,
   f(10)-f(2),
   SQRT(100) +POW(2,3),
   (SELECT tabid FROM systables WHERE tabname = 'sysusers'),
   'T'::BOOLEAN::INT}
);
SELECT * FROM tab1 WHERE
   x=SET{10,
         1+2+3,
         f(10)-f(2),
         SQRT(100) +POW(2,3),
   (SELECT tabid FROM systables WHERE tabname = 'sysusers'),
   'T'::BOOLEAN::INT}
};
```

This assumes that a cast from BOOLEAN to INT exists. (For a more restrictive syntax to specify collection values , see Literal Collection on page 1195.)

NULL Keyword

The NULL keyword is valid in most contexts where you can specify a value. What it specifies, however, is the absence of any value (or an unknown or missing value).

NULL Keyword

"NULL"

Within SQL, the keyword NULL is the only syntactic mechanism for accessing a NULL value. NULL is not equivalent to zero, nor to any specific value. In ascending ORDER BY operations, NULL values precede any non-NULL value; in descending sorts, NULL values follow any non-NULL value. In GROUP BY operations, all NULL values are grouped together. (Such groups might in fact be logically heterogeneous, if they include missing or unknown values.)

The keyword NULL is a global symbol in the syntactic context of expressions, meaning that its scope of reference is global.

Every data type, whether built-in or user-defined, can represent a NULL value. HCL Informix® supports cast expressions in the projection list. This means that users can write expressions of the form NULL**::***datatype*, in which *datatype* is any data type known to the database server.

HCL Informix® supports the typed NULL keyword in general expressions. NULL alone in these scenarios results in a -201 syntax error. As a result, if null is defined as a column name or a procedure name, it must be referenced with a table alias. Otherwise, it returns a -201 syntax error. The behavior is summarized in the following examples and results:

```
create table tab1 (a int, null int);
create table tab2 (a int, b int);
```

Table 20. NULL behavior

Statement	Result
`select null from tab1 where a = 1`	-201 syntax error
`select * from tab1 where null = a`	-201 syntax error
`select * from tab1 where tab1.null = a`	Valid syntax
`select * from tab1 where a = null`	-201 syntax error
`select * from tab2 where a = null`	-201 syntax error
`select * from tab2 where null = a`	-201 syntax error

Table 20. NULL behavior (continued)

Statement	Result
`select * from tab2 where null = a`	-201 syntax error
`select NULL::int from tab1`	Valid syntax
`select NULL::int from tab1`	Valid syntax
`select 1 + NULL::int from tab1`	Valid syntax
`select 1 + NULL::int from tab2`	Valid syntax
`select NULL::int + 1 from tab1`	Valid syntax

HCL Informix® prohibits the redefinition of NULL, because allowing such definition would restrict the global scope of the NULL keyword. For this reason, any mechanism that restricts the global scope or redefines the scope of the keyword NULL will syntactically disable any cast expression involving a NULL value. You must ensure that the occurrence of the keyword NULL receives its global scope in all expression contexts.

For example, consider the following SQL code:

```
CREATE TABLE newtable
(
null int
);

SELECT null, null::int FROM newtable;
```

The CREATE TABLE statement is valid, because the column identifiers have a scope of reference that is restricted to the table definition; they can be accessed only within the scope of a table.

The SELECT statement in the example, however, poses some syntactic ambiguities. Does the identifier **null** appearing in the projection list refer to the global keyword NULL, or does it refer to the column identifier **null** that was declared in the CREATE TABLE statement?

- If the identifier **null** is interpreted as the column name, the global scope of cast expressions with the NULL keyword will be restricted.
- If the identifier **null** is interpreted as the NULL keyword, the SELECT statement must generate a syntactic error for the first occurrence of **null** because the NULL keyword can appear only as a cast expression in the projection list.

A SELECT statement of the following form is valid because the NULL column of **newtable** is qualified with the table name:

```
SELECT newtable.null, null::int FROM newtable;
```

More involved syntactic ambiguities arise in the context of an SPL routine that has a variable named **null**. An example follows:

```
CREATE FUNCTION nulltest() RETURNING INT;
   DEFINE a INT;
   DEFINE null INT;
   DEFINE b INT;
   LET a = 5;
   LET null = 7;
   LET b = null;
   RETURN b;
END FUNCTION;

EXECUTE FUNCTION nulltest();
```

When the preceding function executes in DB-Access, in the expressions of the LET statement, the identifier **null** is treated as the keyword NULL. The function returns a NULL value instead of 7.

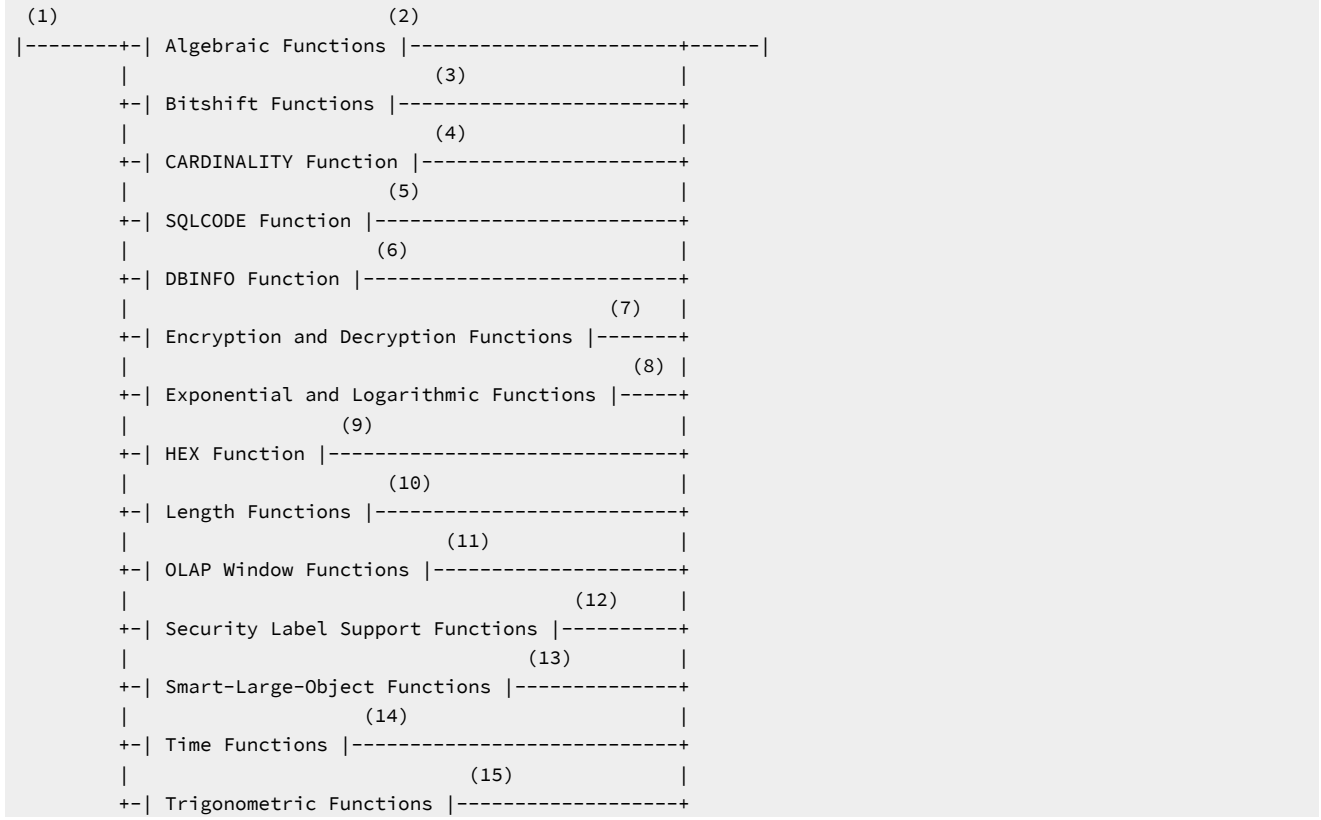
Using **null** as a variable of an SPL routine would restrict the use of a NULL value in the body of the SPL routine. Therefore, the preceding SPL code is not valid, and causes HCL Informix® to return the following error:

```
-947   Declaration of an SPL variable named 'null' conflicts
       with SQL NULL value.
```

In ESQL/C, you should use an indicator variable if there is the possibility that a SELECT statement will return a NULL value.

Function Expressions

A function expression can return one or more values from built-in SQL functions or from user-defined functions, as the following diagram shows.



```
        |                                  (16)        |
        +-| String-Manipulation Functions |-------------+
        |                                (17)          |
        +-| IFX_ALLOW_NEWLINE Function |----------------+
        |                             (18)             |
        '-| User-Defined Functions |--------------------'
```

Notes:

1. Informix® extension
2. See Algebraic Functions on page 1057
3. See Bitshift Functions on page 1069
4. See CARDINALITY Function on page 1070
5. See SQLCODE Function (SPL) on page 1071
6. See DBINFO Function on page 1071
7. See Encryption and decryption functions on page 1079
8. See Exponential and Logarithmic Functions on page 1088
9. See HEX Function on page 1090
10. See Length functions on page 1090
11. See OLAP window expressions on page 1168
12. See Security Label Support Functions on page 1092
13. See Smart-Large-Object Functions on page 1095
14. See Time Functions on page 1101
15. See Trigonometric Functions on page 1108
16. See String-Manipulation Functions on page 1113
17. See IFX_ALLOW_NEWLINE Function on page 1148
18. See User-Defined Functions on page 1149

The following examples show function expressions:

```
EXTEND (call_dtime, YEAR TO SECOND)
```

```
HEX (LENGTH(123))
```

```
MDY (12, 7, 1900 + cur_yr)
```

```
TAN (radians)
```

```
DATE (365/2)
```

```
ABS (-32)
```

```
LENGTH ('abc') + LENGTH (pvar)
```

```
EXP (3)
```

```
HEX (customer_num)
```

```
MOD (10,3)
```

Algebraic Functions

Algebraic functions take one or more arguments of numeric data types. Besides supporting numeric arguments, the **CEIL** and **FLOOR** functions can also take character string arguments that can be converted to DECIMAL values, and the **ROUND** and **TRUNC** functions can also take DATE or DATETIME arguments.

Algebraic Functions

```
{{ ABS | CEIL | FLOOR }(num_expression) | GREATEST( expression ,expression+--) | LEAST( expression
,expression+--) | MOD(dividend, divisor ) | { POW | POWER }(base, exponent) | ROOT(radicand [{ , 2 | ,
index }] ) | { ROUND | TRUNC } ( { num_expression | date_expression } [{ , 0 | , factor }] ) | { ROUND | TRUNC }
(date_expression [ , ' { DD | DAY | MONTH | YEAR } ' ] ) | { ROUND | TRUNC }(datetime_expression [{
, 'DD' | , ' { MI | HH | DAY | MONTH | YEAR } ' }] ) | SQRT( sqrt_radicand) }
```

Element	Description	Restrictions	Syntax
base	Value to be raised to the power specified in *exponent*	Must return a real number	Expression on page 1003
date_expression	Expression that evaluates to (or is cast to) a DATE value	Must return a DATE value	Expression on page 1003
datetime_expression	Expression that evaluates to (or is cast to) a DATETIME value	Must return a DATETIME value	Expression on page 1003
dividend	Value to be divided by *divisor*	A real number	Expression on page 1003
divisor	Value by which to divide *dividend*	A nonzero real number	Expression on page 1003
exponent	Power to which to raise *base*	A real number	Expression on page 1003
factor	Number of significant digits to replace with zero in the returned value. Default is to return the rounded or truncated integer part of the first argument.	Integer in range +32 to -32. Positive or unsigned values are applied to the right of the decimal point, and negative values are applied to the left.	Literal Number on page 1198
index	Root to extract. The default is 2.	A nonzero real number	Expression on page 1003
num_expression	Expression that evaluates to (or is cast to) a numeric value	A real number	Expression on page 1003

Element	Description	Restrictions	Syntax
radicand	Value whose root is to be returned	A real number	Expression on page 1003
sqrt_radicand	Number with a real square root	A nonnegative real number	Expression on page 1003

ABS Function

The ABS function returns the absolute value of its numeric argument, returning the same data type as its argument. The query in the following example returns all orders for which a **ship_charge** greater than $20 was paid in cash (**+**) or as store credit (**-**).

```
SELECT order_num, customer_num, ship_charge
  FROM orders WHERE ABS(ship_charge) > 20;
```

CEIL Function

The **CEIL** function takes as its argument a numeric expression, or a string that can be converted to a DECIMAL data type, and returns the DECIMAL(32) representation of the smallest integer that is greater than or equal to its single argument.

The following query returns `33` as the smallest integer that is larger than or equal to the **CEIL** argument of `32.3`:

```
SELECT CEIL(32.3) FROM systables WHERE tabid = 1;
```

The next example returns `-32` as the smallest integer that is larger than or equal to the **CEIL** argument of `-32.3` :

```
SELECT CEIL(-32.3) FROM systables WHERE tabid = 1;
```

FLOOR Function

The **FLOOR** function takes as its argument a numeric expression, or a string that can be converted to a DECIMAL data type, and returns the DECIMAL(32) representation of the largest integer that is smaller than or equal to its single argument.

The following query returns `32` as the largest integer that is smaller than or equal to the **FLOOR** argument of `32.3`:

```
SELECT FLOOR(32.3) FROM systables WHERE tabid = 1;
```

The next example returns `-33` as the largest integer that is smaller than or equal to the **FLOOR** argument of `-32.3` :

```
SELECT FLOOR(-32.3) FROM systables WHERE tabid = 1;
```

These examples illustrate how the **FLOOR** and **CEIL** functions provide upper and lower bounds that differ by 1 when they have the same argument that has a nonzero fractional part. For an integer argument, **FLOOR** and **CEIL** return the same DECIMAL(32) representation of their argument.

GREATEST function

The **GREATEST** function returns the maximum value in a list of expressions.

The arguments to this function must be comma-separated expressions that evaluate to compatible data types.

This is the syntax of the **GREATEST** function:

GREATEST Function

" **GREATEST**(*expression*, *expression*) "

Element	Description	Restrictions	Syntax
expression	Expression whose value can be compared	Data type cannot be a collection or a large object.	Expression on page 1003

The arguments must be of compatible data types. Arguments of complex data types, or BYTE, TEXT, BLOB, CLOB objects, or DISTINCT types based on any of these data types are not supported. Any user-defined data type that you specify as an argument to the **GREATEST** function must implement the **greaterthan( )** function.

The database server converts the specified *expression* arguments, if necessary, to the data type of the returned value. This return data type is determined by all the operands of the *expression*, and the compatibility rule is consistent with CASE expressions.

The return value of the **GREATEST** function is its largest argument value. If one or more arguments evaluates to NULL, the result is NULL. If **GREATEST** is used to compare DATE or DATETIME values, the return value is the latest date.

Example

Assume that table **T1** contains three columns **C1**, **C2**, and **C3** with values `1`, `7`, and `4`. The following query returns a value of `7`:

```
SELECT GREATEST (C1, C2, C3) FROM T1;
```

If column **C3** has a value of NULL instead of `4`, however, the same query returns a NULL value.

LEAST function

The LEAST function returns the minimum value in a set of values.

LEAST Function

" **LEAST**(, *expression expression*) "

The arguments must be compatible and each argument must be an expression that returns a value of any data type other than complex types, BYTES, TEXT, BLOB, CLOB, or a user-defined type based on any of these types. The user-defined type must implement the support function **lessthan()** in order to use the LEAST function. The selected argument is converted, if necessary, to the data type of the result. The result data type is determined by all the operands and the compatibility rule is consistent with CASE expression.

The result of the function is the smallest argument value. If at least one argument can be null, the result is null. If LEAST is used to compare dates, the return value is the earliest date.

Example

Assume that table T1 contains three columns C1, C2, and C3 with values 1, 7, and 4. The query returns a value of 1.

```
SELECT LEAST (C1, C2, C3) FROM T1
```

If column **C3** has a value of NULL instead of 4, the same query returns a NULL value.

MOD Function

The **MOD** function takes as arguments two real number operands, and returns the remainder from integer division of the integer part of the first argument (the *dividend*) by the integer part of the second argument (the *divisor*). The value returned is an INT data type (or INT8 for remainders outside the range of INT). The quotient and any fractional part of the remainder are discarded. The *divisor* cannot be 0. Thus, `MOD (x,y)` returns `y (modulo x)`. Make sure that any variable that receives the result is of a data type that can store the returned value.

This example tests to see if the current date is within a 30-day billing cycle:

```
SELECT MOD(TODAY - MDY(1,1,YEAR(TODAY)),30) FROM orders;
```

POW Function

The **POW** function raises its first numeric argument, the *base*, to the power of its second numeric argument, the *exponent*. The returned value is a FLOAT data type.

The following example returns all rows from the **circles** table in which the **radius** column value implies an area less than 1,000 square units, using an approximation to pi with a scale of 4:

```
SELECT * FROM circles WHERE (3.1416 * POW(radius,2)) < 1000;
```

The function identifier **Power®** is a synonym for **POW**.

To use *e*, the base of natural logarithms, see EXP Function on page 1088.

ROOT Function

The **ROOT** function extracts a positive real root value, returned as a FLOAT data type, from its first numeric expression argument, the *radicand*.

If you specify a second numeric argument as the *index*, which cannot be zero, then the returned value to the power *index* is equal (within rounding error) to the *radicand* argument. If only the *radicand* argument is supplied, `2` is the default *index* value. You cannot specify zero as the value of *index*.

The first SELECT statement in the following example, which uses the default *index* value of 2, returns the positive square root of the literal number 9. The second example returns the cube root of the literal number 64.

```
SELECT ROOT(9) FROM angles;         -- square root of 9
SELECT ROOT(64,3) FROM angles;      -- cube root of 64
```

Invoking **ROOT** with only a single argument is equivalent to invoking the **SQRT** function.

SQRT Function

The **SQRT** function returns the positive square root of its argument, which must be a non-negative numeric expression.

The following example returns the square root of 9 for each row of the **angles** table:

```
SELECT SQRT(9) FROM angles;
```

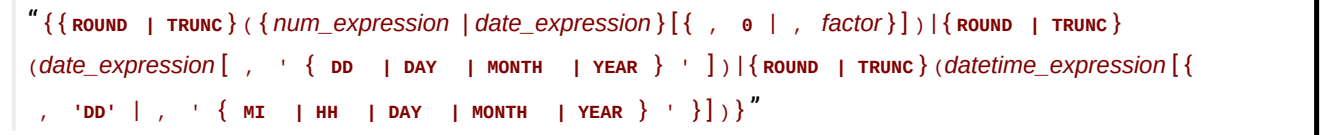
The **SQRT** function is equivalent to `ROOT(x)`, where 2 is the default value of the second argument to the **ROOT** function, specifying the index.

ROUND Function

The **ROUND** function can reduce the precision of its first numeric, MONEY, DATE, or DATETIME argument, and returns the rounded value. If the first argument is not a number, a MONEY value, or a point in time, it must be cast to a numeric, MONEY, DATE, or DATETIME data type.

The following diagram shows the syntax of both the **ROUND** and **TRUNC** algebraic functions, which support the same syntax. Because their semantics differ, however, they can return different values from the same argument list. Only **ROUND** can return an absolute value larger than its first argument.

ROUND and TRUNC algebraic functions



Element	Description	Restrictions	Syntax
date_expression	Expression that evaluates to (or is cast to) a DATE value	Must return a DATE value	Expression on page 1003
datetime_expression	Expression that evaluates to (or is cast to) a DATETIME value	Must return a DATETIME value	Expression on page 1003
factor	Number of significant digits to replace with zero in the returned value. Default is to return the rounded or truncated integer part of the first argument.	Integer in range +32 to -32. Positive or unsigned values are applied to the right of the decimal point, and negative values are applied to the left.	Literal Number on page 1198
num_expression	Expression that evaluates to (or is cast to) a numeric value	A real number	Expression on page 1003

Usage

The **ROUND** function resembles the **TRUNC** function, whose syntax is also shown above.t **ROUND** differs, however, in how it treats any portion of its first argument that is smaller than the least significant digit or time unit within the precision that its explicit or default second argument specifies.

- If the absolute value of this portion is equal to or greater than half of the smallest unit within the precision, the value of that digit or time unit is incremented by 1 in the value returned by **ROUND**. If this portion is less than half of a unit, however, it is discarded, and only the digits or time units of the first argument within the specified or default precision are returned.

  That is, if the first argument is greater than zero,
  - the **ROUND** function rounds down any portion of its first argument that is smaller than half a unit of the least significant digit or time unit within the precision of the second argument,
  - but any portion of the first argument that is equal to or greater than half a unit is rounded up.

  For example, `ROUND(3.5,0) = 4` and `ROUND(3.4,0) = 3`.

  But if the first argument is less than zero,
  - the **ROUND** function rounds up any portion of its first argument that is smaller than half a unit of the least significant digit or time unit within the precision of the second argument,
  - but any portion of the first argument that is equal to or greater than half a unit is rounded down.

  For example, `ROUND(-3.5,0) = -4` and `ROUND(-3.4,0) = -3`.
- The **TRUNC** function, in contrast, replaces with zero any digits less than the specified precision for numeric expressions. For DATE or DATETIME expressions, **TRUNC** freplaces any time units smaller than the specified format string with 1 for *month* or *day* time units, or with zero for time units smaller than *day*.

The **ROUND** function can accept an optional second argument that specifies the precision of the returned value. The syntax and semantics of the second argument depend on whether the first argument is a number expression, a DATETIME expression, or DATE expression.

Rounding numeric and MONEY values

- When the first argument is a numeric expression, the returned value is a DECIMAL and the second argument can be an integer in the range from -32 to +32 inclusive, specifying the position (relative to the decimal point) of the last significant digit of the returned value. If you omit the *factor* specification when the first argument is numeric, **ROUND** returns the integer value of the first argument rounded to a scale of zero, or to the units place.

  Positive-digit values specify rounding to the right of the decimal point; negative-digit values specify rounding to the left of the decimal point, as Figure 12: Examples of negative, zero, and positive rounding factors on page 1063 shows:

Figure 12. Examples of negative, zero, and positive rounding factors



The following example uses the **ROUND** function with a column expression as its first argument and no second argument, so that the numeric expression is rounded to a scale of zero. This query returns the order number and rounded total price of items whose total price (rounded to the default scale of zero decimal places) is equal to $124.00.

```
SELECT order_num , ROUND(total_price) FROM items
   WHERE ROUND(total_price) = 124.00;
```

If you use a MONEY data type as the argument for the **ROUND** function and you round to an explicit or default scale of zero, the returned value is represented with `.00` as the fractional part. The SELECT statement in the following example rounds 125.46 and a MONEY column value. The query returns 125 and a rounded price in the form `xxx.00` for each row in the **items** table.

```
SELECT ROUND(125.46), ROUND(total_price) FROM items;
```

Rounding DATE and DATETIME values

- When the first argument to **ROUND** is a DATETIME expression, the returned value is a DATETIME YEAR TO MINUTE data type and the second argument must be a quoted string that specifies the smallest significant time unit in the returned value. If you omit the second argument, the default format string is `'DD'`, specifying the nearest day, with the hour and minute rounded to `00:00`. The following format strings are valid as the second argument:

Table 21. Format strings for DATETIME arguments to the ROUND function

Format String	Effect on Returned DATETIME Value
'`YEAR`'	Rounded to the beginning of the nearest year, with dates after June 30 rounded up to the next year. The *month*, *day*, *hour*, and *minute* values round to `-01-01 00:00`.
'`MONTH`'	Rounded to the beginning of the nearest month. Dates after the 15th are rounded up to the next month. The *day*, *hour*, and *minute* values round to `01 00:00`.
'`DD`'	Rounded to the beginning (`00:00` = midnight) of the nearest day. DATETIME values later than 12:00 noon are rounded up to the next day.
'`DAY`'	Rounded to the beginning of the nearest Sunday. Dates that fall on Wednesday, Thursday, Friday, or Saturday are rounded up to the next Sunday.
'`HH`'	Rounded to the beginning of the nearest hour. Time of day values with *minute*:*second* later than `29:59` are rounded up to the next hour. Minutes round to zero.
'`MI`'	Rounded to the beginning of the nearest minute. Time of day values with *second* later than `30` are rounded up to the next minute.

If you omit the format string specification after an initial DATETIME expression argument, the returned value is the value of the first argument rounded to the nearest day, as if you had specified 'DD' as the format string.

Examples that follow use the **ROUND** function with a column expression that returns a DATETIME YEAR TO FRACTION(5) value in a SELECT statement. In these queries, table **mytab** has only a single row, and in that row the value of **mytab.col_dt** is `2012-12-07 14:30:12.12300`.

The following query specifies `'YEAR'` as the DATETIME format string:

```
SELECT ROUND(col_dt, 'YEAR') FROM mytab;
```

The value returned is `2013-01-01 00:00`.

The next query resembles the previous query, but casts the returned value to a DATE data type:

```
SELECT ROUND(col_dt, 'YEAR')::DATE FROM mytab;
```

The value returned is `01/01/2013`.

This example specifies `'MONTH'` as the DATETIME format string:

```
SELECT ROUND(col_dt, 'MONTH') FROM mytab;
```

The value returned is `2012-12-01 00:00`.

This example rounds the DATETIME expression to YEAR TO HOUR precision:

```
SELECT ROUND(col_dt, 'HH') FROM mytab;
```

The value returned is `2012-12-07 15:00`.

- When the first argument is a DATE expression, the returned value is also a DATE data type if the second argument is a quoted string that specifies the smallest time unit in the returned value. These are the same format strings as for rounding DATETIME values, except that `'HH'` and `'MI'` are not valid for DATE values. There is no default format string for rounding DATE arguments.

To return formatted DATE values, you must specify one of the following quoted strings as the second argument to the **ROUND** function:

Table 22. Format strings for DATE arguments to the ROUND function

Format String	Effect on Returned DATE Value
'`YEAR`'	Rounded to the beginning of the nearest year. Dates after June 30 are rounded up to the next year. The *month* and *day* values each round to `01`.
'`MONTH`'	Rounded to the beginning of the nearest month. Dates after the 15th are rounded up to the next month. The returned *day* value is `01`.
'`DD`'	The DATE value of the first *date_expression* argument is returned.
'`DAY`'	The value is rounded to the nearest Sunday. If the first argument is a Sunday, that date is returned. Dates that fall on Wednesday, Thursday, Friday, or Saturday are rounded up to the next Sunday.

If you specify no format string as the second argument when the first argument is a DATE data type, no format string takes effect as the default. No error is issued, but the first argument is treated as numeric expression that evaluates to an integer, rather than as a DATE value. Informix® stores DATE values internally as the integer count of days since 31 December 1899. For dates in the 21st century, integer equivalents to DATE values are 5-digit integers, ranging between approximately 37,000 and 74,000.

For example, the query `SELECT ROUND(TODAY) FROM systables` provides no format string for a DATE expression, and returns the integer `40999` if the query is issued on 1 April 2012.

If you apply a numeric format specification as the second argument, nonnegative numbers have no effect on DATE values, but the following example rounds the last two digits of the returned value to zero:

```
SELECT ROUND(TODAY, -2) FROM systables;
```

On 1 April 2012, the query above would return the integer value `40900`.

On the next day, 2 April 2012, the same query would return the integer value `41000`.

For applications where integer-format dates like `41000` are unhelpful, you can use the 'YEAR', 'MONTH', 'DAY', or 'DD' format strings as the second argument to the **ROUND** function to prevent the DATE argument from being processed as if it were a number expression. On 1 April 2012, the following query returns the DATE value `04/01/2012` if `MDY4/` is the **DBDATE** environment variable setting:

```
SELECT ROUND(TODAY, 'DD') FROM systables WHERE tabid = 1;
```

In the following example, a query is issued on Tuesday, April 3, 2012:

```
SELECT ROUND(TODAY, 'DAY') FROM mytab;
```

The returned value is `03/31/2012`, the current date rounded to the nearest Sunday.

If you are using a host variable to store a rounded point-in-time value in dynamic SQL, and the data type of the first argument is not known at prepare time, Informix® assumes that a DATETIME data type is the first argument to the **ROUND** function and returns a DATETIME YEAR TO MINUTE rounded value. At execution time, after the statement is prepared, error -9750 is issued if a DATE value is supplied for the host variable. To prevent this error, you can specify the data type for the host variable by using a cast, as in this program fragment.

```
sprintf(query1, ",
   "select round( ?::date, 'DAY') from mytab");
EXEC SQL prepare selectq from :query;
EXEC SQL declare select_cursor cursor for selectq;
EXEC SQL open select_cursor using :hostvar_date_input;

EXEC SQL fetch select_cursor into :var_date_output;
```

For the order of precedence among the Informix® environment variables that can specify the display and data entry formats for the built-in chronological data types, see the topic Precedence of DATE and DATETIME format specifications.

TRUNC Function

The **TRUNC** function can reduce the precision of its first numeric, DATE, or DATETIME argument by returning the truncated value. If the first argument is neither a number nor a point in time, it must be cast to a numeric, DATE, or DATETIME data type.

The **TRUNC** function can reduce the precision of its first numeric, DATE, or DATETIME argument by returning the truncated value. If the first argument is neither a number nor a point in time, it must be cast to a numeric, DATE, or DATETIME data type.

The **TRUNC** function resembles the **ROUND** function, but truncates (rather than rounds to the nearest whole number) any portion of its first argument that is smaller than the least significant digit or time unit within the precision that its second argument specifies.

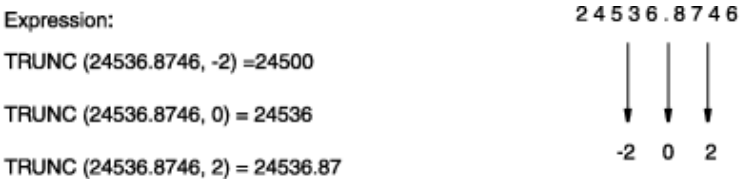
- For numeric expressions, **TRUNC** replaces with zero any digits less than the specified precision.
- For DATE or DATETIME expressions, **TRUNC** replaces any time units smaller than the format specification with `1` for *month* or *day* time units, or with `0` for time units smaller than *day*.

The **TRUNC** function can accept an optional second argument that specifies the precision of the returned value.

- When the first argument is a numeric expression, the second argument must be an integer in the range from -32 to +32 inclusive, specifying the position (relative to the decimal point) of the last significant digit of the returned value. If you omit the *factor* specification when the first argument is numeric, **TRUNC** returns the value of the first argument truncated to a scale of zero, or to the units place.

  Positive digit values specify truncation to the right of the decimal point; negative digit values specify truncation to the left, as Figure 13: Examples of negative, zero, and positive truncation factors on page 1066 shows.

  Figure 13. Examples of negative, zero, and positive truncation factors

  The following example calls the **TRUNC** function with a column expression that returns a numeric value in a SELECT statement. This statement displays the order number and truncated total price of items whose total price (truncated to the default scale of zero decimal places) is equal to $124.00.

  ```
  SELECT order_num , TRUNC(total_price) FROM items
     WHERE TRUNC(total_price) = 124.00;
  ```

  If a MONEY data type is the argument in a call to the **TRUNC** function that specifies a scale of zero, the fractional part becomes `.00` in the returned value. For example, the following SELECT statement truncates 125.46 and a MONEY column value. It returns 125 and a truncated price in the form `xxx.00` for each row in the **items** table.

  ```
  SELECT TRUNC(125.46), TRUNC(total_price) FROM items;
  ```

- When the first argument to **TRUNC** is a DATETIME expression, the second argument must be a quoted string that specifies the smallest significant time unit in the returned value. Only the following format strings are valid as the second argument:

Table 23. Format strings for DATETIME arguments to the TRUNC function

Format String	Effect on Returned Value
'YEAR'	Truncated to the beginning of the year. The *month*, *day*, *hour*, and *minute* values truncate to 01-01 00:00.
'MONTH'	Truncated to the beginning of the first day of the month. The *hour* and *minute* values round to 00:00.
'DD'	Truncated to the beginning (00:00 = midnight) of the same day.
'DAY'	If the first argument is a Sunday, midnight (00:00) on that date is returned. For any other day of the week, midnight on the previous Sunday is returned.
'HH'	Truncated to the beginning of the hour. The *minute* value truncates to zero.
'MI'	Truncated to the beginning of the nearest minute. As for all of these format strings, time units smaller than *minute* are discarded.

If you omit the format string specification after an initial DATETIME expression argument, the returned value is the value of the first argument truncated to the day, as if you had specified 'DD' as the format string.

Examples that follow invoke the **TRUNC** function with a column expression that returns a DATETIME YEAR TO FRACTION(5) value in a SELECT statement. In these examples, table **mytab** has only a single row, and in that row the value of **mytab.col_dt** is 2006-12-07 14:30:12.12300.

This query specifies 'YEAR' as the DATETIME format string:

```
SELECT TRUNC(col_dt, 'YEAR') FROM mytab;
```

The value returned is 2006-01-01 00:00.

The next query resembles the previous query, but casts the truncated value to a DATE data type:

```
SELECT TRUNC(col_dt, 'YEAR')::DATE FROM mytab;
```

The value returned is 01/01/2006.

This example specifies 'MONTH' as the DATETIME format string:

```
SELECT TRUNC(col_dt, 'MONTH') FROM mytab;
```

The value returned is 2006-12-01 00:00.

The following example truncates the DATETIME expression to YEAR TO HOUR precision:

```
SELECT TRUNC(col_dt, 'HH') FROM mytab;
```

The value returned is 2006-12-07 14:00.

- When the first argument is a DATE expression, the second argument should generally be a quoted string that specifies the smallest time unit in the returned value. These are the same format strings as for truncating DATETIME values, except that `'HH'` and `'MI'` are not valid for dates, and there is no default format string for truncating DATE expression arguments.

  To return formatted DATE values, you must use one of the following quoted strings as the second argument to the **TRUNC** function:

  Table 24. Format strings for DATE arguments to the TRUNC function

Format String	Effect on Returned Value
'YEAR'	Truncated to the beginning of the year. The *month* and *day* values are each `01`.
'MONTH'	Truncated to the beginning of the month. The *day* value is `01`.
'DD'	The DATE value of the first *date_expression* argument is returned.
'DAY'	If the first argument is a Sunday, that date is returned. For any other day of the week, the date of the previous Sunday is returned.

If you specify no format string as the second argument when the first argument is a DATE data type, no format string takes effect as the default. No error is issued, but the first argument is treated as numeric expression that evaluates to an integer, rather than as a DATE value. Informix® stores DATE values internally as the integer count of days since 31 December 1899.

For example, the query `SELECT ROUND(TODAY) FROM systables` provides no format string for a DATE expression, and returns the integer `39538` if the query is issued on 1 April 2008.

If you apply a numeric format specification as the second argument, nonnegative numbers have no effect on DATE values, but the following example rounds the last two digits of the returned value to zero:

```
SELECT TRUNC(TODAY, -2) FROM systables;
```

For applications where integer dates like `39500` are unhelpful, use the `'YEAR'`, `'MONTH'`, `'DAY'`, or `'DD'` format strings as the second argument to the **TRUNC** function, to prevent the DATE expression from being processed as if it were a number expression. On 1 April 2008, the following query returns the DATE value `04/01/2008` if `MDY4/` is the setting of the **DBDATE** environment variable:

```
SELECT TRUNC(TODAY, 'DD') FROM systables;
```

If you are using a host variable to store a truncated point-in-time value in dynamic SQL, and the data type of the first argument is not known at prepare time, Informix® assumes that a DATETIME data type is the first argument to the **TRUNC** function and returns a DATETIME YEAR TO MINUTE truncated value. At execution time, after the statement is prepared, error -9750 is issued if a DATE value is supplied for the host variable. To prevent this error, you can specify the data type for the host variable by using a cast, as in this program fragment.

```
sprintf(query2, "%s",
   "select trunc( ?::date, 'DAY') from mytab");
EXEC SQL prepare selectq from :query2;
EXEC SQL declare select_cursor cursor for selectq;
EXEC SQL open select_cursor using :hostvar_date_input;
```

```
EXEC SQL fetch select_cursor into :var_date_output;
```

For the order of precedence among the Informix® environment variables that can specify the display and data entry formats for the built-in chronological data types, see the topic Precedence of DATE and DATETIME format specifications.

Note that the **TRUNC** function name is based on a use of the English word "truncate" that is different from its meaning in the TRUNCATE statement of SQL. The **TRUNC** function replaces the value of its first argument with another value that has a smaller precision or the same precision. The TRUNCATE statement deletes all of the rows from a database table, without dropping the table schema.

Bitshift Functions

Bitshift functions perform a bit shift operation on an integer or bigint value and return the shift result.

The bitshift functions have the following syntax.

```
|--+-IFX_BIT_LEFTSHIFT---+--(--operand--,--amount--)--|
   '-IFX_BIT_RIGHTSHIFT--'
```

Element	Description	Restrictions	Syntax
operand	Entity whose bits are to be shifted. Its type will also be function's return type.	Must be of type integer or bigint.	Expression on page 1003
amount	Number of bits operand will be shifted by.	Integer value between 0 and number of bits in operand's type (or respective modulo of any other integer value).	Expression on page 1003

IFX_BIT_LEFTSHIFT Function

The **IFX_BIT_LEFTSHIFT** function performs a bit left shift operation on an integer or bigint value and returns the shift result

The IFX_BIT_LEFTSHIFT function performs a bit left shift operation on an integer or bigint value and returns the shift result. It takes as arguments an integer or bigint operand and an integer operator. The type of the operand will be the function's return type. The operator determines the number of bits by which the passed in value gets shifted to the left, its value should range between 0 and 31 for integer operands, and between 0 and 63 for bigint operands (any other values might be taken as their modulo of 32 or 64 respectively, but that's not guaranteed).

The effect of ifx_bit_leftshift is equivalent to multiplying the operand by 2 to the power of operator, but is of course limited to the resulting type's capacity.

With both integer and bigint, the value with only the high bit set carries special meaning as it signifies a NULL value. Should the bit shift result in an integer 0x80000000, you'd receive error 1215, whereas a bigint result 0x8000000000000000 would be returned as NULL.

```
Examples:
```

```
          operand    operator      result
                1           4          16  (0x00000010)
                3           2          12  (0x0000000c)
  "0x800"::integer           5       65536  (0x00010000)
            12345          26  -469762048  (0xe4000000)
                1          31  error 1215
        1::bigint          31  2147483648  (0x80000000)
```

IFX_BIT_RIGHTSHIFT Function

The **IFX_BIT_RIGHTSHIFT** function performs a bit right shift operation on an integer or bigint value and returns the shift result.

The IFX_BIT_RIGHTSHIFT function performs a bit right shift operation on an integer or bigint value and returns the shift result. It takes as arguments an integer or bigint operand and an integer operator. The type of the operand will be the function's return type. The operator determines the number of bits by which the passed in value gets shifted to the right, its value should range between 0 and 31 for integer operands, and between 0 and 63 for bigint operands (any other values might be taken as their modulo of 32 or 64 respectively, but that's not guaranteed).

```
Example:

select name, IFX_BIT_RIGHTSHIFT(partnum, 20) as dbspace_num
  from sysmaster:sysdatabases;
```

CARDINALITY Function

The **CARDINALITY** function returns the number of elements in a collection column (SET, MULTISET, LIST).

The **CARDINALITY** function has the following syntax.

CARDINALITY Function

" CARDINALITY "

" ({ *collection_col* | *collection_var* }) "

Element	Description	Restrictions	Syntax
collection_col	An existing collection column	Must be declared as a collection data type	Identifier on page 1231
collection_var	Host or program collection variable	Must be declared as a collection data type	Language specific

Suppose that the **set_col** SET column contains the following value:

```
{3, 7, 9, 16, 0}
```

The following SELECT statement returns 5 as the number of elements in the **set_col** column:

```
SELECT CARDINALITY(set_col)
   FROM table1;
```

If the collection contains duplicate elements, **CARDINALITY** counts each individual element.

SQLCODE Function (SPL)

The **SQLCODE** function takes no arguments, but returns to its calling context the value of **sqlca.sqlcode** for the most recently executed SQL statement (whether static or dynamic) that the current SPL routine has executed. Only use **SQLCODE** in the context of a cursor.

SQLCODE

> " SQLCODE "

You can use **SQLCODE** in expressions within SPL routines to identify the state of a dynamic cursor. This built-in function is useful in error handling and in contexts such as determining whether a query or function call has returned no rows, or when a cursor has reached the last row of the active set, or to identify other conditions when SPL program control should exit from a loop.

The following SPL program fragment illustrates the use of **SQLCODE** to detect the end of the active set of a cursor within a WHILE loop.

Example

```
CREATE PROCEDURE ...
...
DEFINE myc1 ...
...
PREPARE p FOR "SELECT c1 FROM t1";
DECLARE cur FROM s;
OPEN cur;

FETCH cur INTO myc1;
WHILE (SQLCODE != 100)
FETCH cur INTO myc1;
        -- process myc1
...
END WHILE;

END PROCEDURE;
```

The **SQLCODE** function is not needed in UDRs written in ESQL/C, which have direct access to the SQL Communications Area (**SQLCA**) through the GET DIAGNOSTICS statement of Dynamic SQL and by other mechanisms. The database server issues an error if the calling context of the built-in **SQLCODE** function is not an SPL routine.

DBINFO Function

The following diagram shows the syntax of the **DBINFO** function.

(explicit id unique_1482_Connect_42_difu001) unique_1482_Connect_42_difu001

DBINFO Function

" DBINFO "

" ({ 'dbspace' , { tblspace_num | expression } | { 'sqlca.sqlerrd1' | 'sqlca.sqlerrd2' } | { | 'sessionid' | 'cdrsession' | 'dbname' | 'dbhostname' | 'serial8' | 'bigserial' | 'get_tz' | 'utc_current' | 'utc_to_datetime' , { utc_value | table.column } | { 'version' , 'parameter' } } }) "

Element	Description	Restrictions	Syntax
column	Name of a column in the *table*	Must exist in *table*	Identifier on page 1231
expression	Expression that evaluates to *tblspace_num*	Can contain column names, SPL variables, host variables, or subqueries, but must return a numeric value	Expression on page 1003
parameter	Quoted string that specifies which part of the *version* string to return	For valid *parameter* values, see #unique_1495	See the **Restrictions** column.
table	Table for which to display the dbspace name or containing an integer *column* of UTC values.	Must match the name of a table in the FROM clause of the query	Identifier on page 1231
tblspace_num	Tblspace number (partition number) of a table	Must exist in the **partnum** column of the **systables** table for the database	Literal Number on page 1198
utc_value	A UTC value to be converted to the DATETIME equivalent	Must be a numeric expression that evaluates to the number of seconds since 1970-01-01 00:00:00+00:00	Expression on page 1003, Literal Number on page 1198

DBINFO Options

The **DBINFO** function is actually a set of functions that return different types of information about the database. To invoke each function, specify its option as the quoted-string argument after the **DBINFO** keyword. You can use any **DBINFO** option anywhere within SQL statements and within UDRs.

The following table shows the categories of database and database server information that Informix® can retrieve with valid **DBINFO** options.

- The **Arguments** column shows the parentheses-delimited argument list of each valid **DBINFO** option.
- The **Information Returned** column shows the type of database information that the **Arguments** option retrieves.
- The **Topic** column shows where you can find more information about the **Arguments** option.

Arguments	Information Returned	Topic
('dbhostname')	The host name of the database server to which a client application is connected	Using the 'dbhostname' Option on page 1077
('dbname')	The identifier of the database to which a client application is connected	#unique_1487
('dbspace' *tblspace_num*)	The name of a dbspace corresponding to a tblspace number	Using the ('dbspace', tblspace_num) Option on page 1074
('get_tz')	The time zone of the session, **$TZ**, as specified as a string by the client.	Using the 'get_tz' Option on page 1078
('serial8')	The last SERIAL8 value inserted in a table	Using the 'serial8' and 'bigserial' options on page 1077
('bigserial')	The last BIGSERIAL value inserted in a table	Using the 'serial8' and 'bigserial' options on page 1077
('sessionid')	The session ID number of the current session	Using the 'sessionid' Option on page 1075
('cdrsession')	Whether a thread is performing an Enterprise Replication operation	Using the 'cdrsession' option on page 1076
('sqlca.sqlerrd1')	The last SERIAL value inserted in a table	Using the 'sqlca.sqlerrd1' Option on page 1074
('sqlca.sqlerrd2')	The number of rows processed by SELECT, INSERT, DELETE, UPDATE, EXECUTE PROCEDURE, and EXECUTE FUNCTION statements	Using the 'sqlca.sqlerrd2' Option on page 1075
('utc_current')	The current UTC time value (as an integer number of seconds since 1970-01-01 00:00:00+00:00) when the SQL statement began to execute.	Using the 'utc_current' Option on page 1078
('utc_to_datetime', *table.column*)	The DATETIME value corresponding to a specified integer column containing a UTC time value (as an integer number of seconds since 1970-01-01 00:00:00+00:00).	Using the 'utc_to_datetime' Option on page 1078

Arguments	Information Returned	Topic
('utc_to_datetime', *utc_value*)	The DATETIME value corresponding to a specified UTC time value (as an integer number of seconds since 1970-01-01 00:00:00+00:00).	Using the 'utc_to_datetime' Option on page 1078
('version', '*parameter*')	Type of the database server and its release version to which the client application is connected. (The call to **DBINFO** fails with an error if no *parameter* specifies a format for the version information.)	#unique_1495

Using the ('dbspace', tblspace_num) Option

The **'dbspace'** option returns a character string that contains the name of the dbspace that corresponds to a tblspace number. You must supply an additional parameter, either *tblspace_num* or an expression that evaluates to *tblspace_num*. The following example uses the **'dbspace'** option. First, it queries the **systables** system catalog table to determine the *tblspace_num* for the table customer, then it executes the function to determine the *dbspace* name.

```
SELECT tabname, partnum FROM systables
   where tabname = 'customer';
```

If the query returns a partition number of 1048892, you insert that value into the second argument to find which dbspace contains the **customer** table, as the following example shows:

```
SELECT DBINFO ('dbspace', 1048892) FROM systables
   where tabname = 'customer';
```

If the table for which you want to know the dbspace name is fragmented, you must query the **sysfragments** system catalog table to find out the tblspace number of each table fragment. Then you must supply each tblspace number in a separate **DBINFO** query to find out all the dbspaces across which a table is fragmented.

Using the 'sqlca.sqlerrd1' Option

The 'sqlca.sqlerrd1' option returns a single integer that provides the last serial value that is inserted into a table. To ensure valid results, use this option immediately following a singleton INSERT statement that inserts a single row with a serial value into a table.

Tip: To obtain the value of the last SERIAL8 value that is inserted into a table, use the 'serial8' option of DBINFO. For more information, see Using the 'serial8' and 'bigserial' options on page 1077.

The following example uses the 'sqlca.sqlerrd1' option:

```
EXEC SQL create table fst_tab (ordernum serial, partnum int);
EXEC SQL create table sec_tab (ordernum serial);
EXEC SQL insert into fst_tab VALUES (0,1);
EXEC SQL insert into fst_tab VALUES (0,4);
```

```
EXEC SQL insert into fst_tab VALUES (0,6);
EXEC SQL insert into sec_tab values (dbinfo('sqlca.sqlerrd1'));
```

This example inserts a row that contains a primary-key serial value into the **fst_tab** table, and then uses the DBINFO function to insert the same serial value into the **sec_tab** table. The value that the DBINFO function returns is the serial value of the last row that is inserted into **fst_tab**.

Because the SQLCA structure does not record serial values that are inserted by triggers, you cannot call the DBINFO function with the 'sqlca.sqlerrd1', 'bigserial', or 'serial8' options to return a serial value that a triggered action inserts.

For more information about the SQL Communications Area (SQLCA) data structure, within which sqlca.sqlerrd1 is a field, see the *HCL® Informix® Guide to SQL: Tutorial*.

Using the 'sqlca.sqlerrd2' Option

The **'sqlca.sqlerrd2'** option returns a single integer that provides the number of rows that SELECT, INSERT, DELETE, UPDATE, EXECUTE PROCEDURE, and EXECUTE FUNCTION statements processed. To ensure valid results, use this option after SELECT, EXECUTE PROCEDURE, and EXECUTE FUNCTION statements have completed executing. In addition, to ensure valid results when you use this option within cursors, make sure that all rows are fetched before the cursors are closed.

The following example shows an SPL routine that uses the **'sqlca.sqlerrd2'** option to determine the number of rows that are deleted from a table:

```
CREATE FUNCTION del_rows (pnumb INT)
RETURNING INT;

DEFINE nrows INT;

DELETE FROM fst_tab WHERE part_number = pnumb;
LET nrows = DBINFO('sqlca.sqlerrd2');
RETURN nrows;

END FUNCTION;
```

For more information about the SQL Communications Area (SQLCA) data structure, within which **sqlca.sqlerrd2** is a field, see the *HCL® Informix® Guide to SQL: Tutorial*.

Using the 'sessionid' Option

The **'sessionid'** option of the **DBINFO** function returns the session ID of your current session. When a client application makes a connection to the database server, the database server starts a session with the client and assigns a session ID for the client. The session ID serves as a unique identifier for a given connection between a client and a database server.

The database server stores the value of the session ID in a data structure in shared memory that is called the *session control block*. The session control block for a given session also includes the user ID, the process ID of the client, the name of the host computer, and a variety of status flags.

When you specify the **'sessionid'** option, the database server retrieves the session ID of your current session from the session control block and returns this value to you as an integer. Some of the System-Monitoring Interface (SMI) tables in the

sysmaster database include a column for session IDs, so you can use the session ID that the **DBINFO** function obtained to extract information about your own session from these SMI tables. For further information on the session control block, see the *HCL® Informix® Administrator's Guide*. For further information on the **sysmaster** database and the SMI tables, see the *HCL® Informix® Administrator's Reference*.

In the following example, the user specifies the **DBINFO** function in a SELECT statement to obtain the value of the current session ID. The user poses this query against the **systables** system catalog table and uses a WHERE clause to limit the query result to a single row.

```
SELECT DBINFO('sessionid') AS my_sessionid
   FROM systables
   WHERE tabname = 'systables';
```

In the preceding example, the SELECT statement queries against the **systables** system catalog table. You can, however, obtain the session ID of the current session by querying against any system catalog table or user table in the database. For example, you can enter the following query to obtain the session ID of your current session:

```
SELECT DBINFO('sessionid') AS user_sessionid
   FROM customer
   WHERE customer_num = 101;
```

You can use the **DBINFO 'sessionid'** option not only in SQL statements but also in SPL routines. The following example shows an SPL function that returns the value of the current session ID to the calling program or routine:

```
CREATE FUNCTION get_sess()
   RETURNING INT;
   RETURN DBINFO('sessionid');
END FUNCTION;
```

Using the 'cdrsession' option

The 'cdrsession' option to the DBINFO() function detects if an INSERT, UPDATE, or DELETE statement is being performed as part of a replicated transaction.

You might want to design triggers, stored procedures, or user-defined routines to take different actions depending on whether a transaction is being performed as part of Enterprise Replication. The 'cdrsession' option of the DBINFO() function returns `1` if the thread performing the database operation is an Enterprise Replication apply or sync thread; otherwise, the function returns `0`.

The following example shows an SPL function that uses the 'cdrsession' option to determine if a thread is performing an Enterprise Replication operation:

```
CREATE FUNCTION iscdr ()
RETURNING int;

DEFINE iscdrthread int;
SELECT DBINFO('cdrsession') into iscdrthread
from systables where tabid = 1;
RETURN iscdrthread;

END FUNCTION
```

Using the 'dbhostname' Option

You can use the **'dbhostname'** option to retrieve the host name of the database server to which a database client is connected.

This option retrieves the physical computer name of the computer on which the database server is running.

In the following example, the user enters the **'dbhostname'** option of **DBINFO** in a SELECT statement to retrieve the host name of the database server to which is connected:

```
SELECT DBINFO('dbhostname')
   FROM systables
   WHERE tabid = 1;
```

The following table shows the result of this query.

(constant)
rd_lab1

Using the 'serial8' and 'bigserial' options

The 'bigserial' and 'serial8' options respectively return a single integer that specifies the last SERIAL8 or BIGSERIAL value that was inserted into a table. To ensure valid results, use this option immediately following an INSERT statement that inserts a SERIAL8 or BIGSERIAL value.

Tip: To obtain the value of the last SERIAL value that is inserted into a table, use the 'sqlca.sqlerrd1' option of DBINFO( ). For more information, see Using the 'sqlca.sqlerrd1' Option on page 1074.

The following example uses the 'serial8' option:

```
EXEC SQL CREATE TABLE fst_tab
   (ordernum SERIAL8, partnum INT);
EXEC SQL CREATE TABLE sec_tab (ordernum SERIAL8);

EXEC SQL INSERT INTO fst_tab VALUES (0,1);
EXEC SQL INSERT INTO fst_tab VALUES (0,4);
EXEC SQL INSERT INTO fst_tab VALUES (0,6);

EXEC SQL INSERT INTO sec_tab
   SELECT dbinfo('serial8')
   FROM fst_tab WHERE partnum = 6;
```

This example inserts a row that contains a primary-key SERIAL8 value into the **fst_tab** table and then uses the DBINFO function to insert the same SERIAL8 value into the **sec_tab** table. The value that the DBINFO function returns is the SERIAL8 value of the last row that is inserted into **fst_tab**. The subquery in the last line contains a WHERE clause so that a single value is returned.

The SQLCA structure does not record serial values that are inserted by triggers. You cannot call the DBINFO function with the 'bigserial' option to return the most recent BIGSERIAL value that was inserted directly by the triggered action of a trigger on a

table (nor of an INSTEAD OF trigger on a view). For the same reason, the DBINFO ('serial8') function cannot return a SERIAL8 value that was inserted by a trigger on a table, nor by an INSTEAD OF trigger on a view.

Using the 'get_tz' Option

The **'get_tz'** option returns the **$TZ** string that shows the time zone of the current session.

The following example uses the **'get_tz'** option in a query of the **cust_calls** table of the **stores_demo** database:

```
EXEC SQL select first call_dtime, dbinfo('get_tz')
   from cust_calls where customer_num = 106;
```

This example returns a string value of the session time zone and the first **call_dtime** value in the **cust_calls** table for which the **customer_num** value is `106`.

Using the 'utc_current' Option

The **'utc_current'** option returns the current value of Coordinated Universal Time (UTC) as an integer value that shows the number of seconds that have elapsed between 1970-01-01 00:00:00+00:00) and when the current SQL statement began to execute.

Unlike Universal Time (UT), which calculates the duration of seconds from the earth's rotation, UTC uses seconds of a fixed length, based on high-precision atomic clocks.

Because of variation in the earth's gradually diminishing rotation rate, intercalary *leap seconds* are introduced from time to time in UTC to reduce discrepancies with UT time. By default, Informix® ignores leap seconds in DATETIME and INTERVAL arithmetic. When Informix® is supported by an operating system that takes leap seconds into account, however, the leap seconds are reflected in subsequent DATETIME and INTERVAL operations after the operating system adjusts the system clock for leap seconds.

Using the 'utc_to_datetime' Option

The 'utc_to_datetime' option of the DBINFO function returns the UTC seconds to DATETIME value that the server would generate if the UNIX™ time( ) system call returned the value of the second parameter, taking into account the time zone of the database server.

The 'utc_to_datetime' option casts to a DATETIME value its last argument, which must be a numeric expression representing a Coordinated Universal Time (UTC) value. If this evaluates to a number with a fractional part, any fractional seconds are ignored.

In the first example below, the last argument is a UTC value represented as a literal integer. In the second example, the last argument is a column expression specifying an integer column that stores UTC values. In both examples, DBINFO casts the UTC value to a DATETIME value in the time zone of the database server:

```
DBINFO ('utc_to_datetime', 1299912999 )

DBINFO ('utc_to_datetime', timesheet.utc_checkin )
```

If the value of the last argument is negative, the function returns a DATETIME value from an earlier UNIX™ epoch, as in the next example:

```
SELECT DBINFO("utc_to_datetime", -2134567890.91234)
   FROM 'sysmaster:"informix".sysdual';
```

This query returns the DATETIME value `1902-05-12 08:28:30`.

These example times all assume that the server is in a specific time zone. The following query returns four DATETIME values:

```
SELECT
    DBINFO('utc_to_datetime', -32767) AS min_smallint,
    DBINFO('utc_to_datetime', +32767) AS max_smallint,
    DBINFO('utc_to_datetime', 1299912999),
    DBINFO("utc_to_datetime", -2134567890.91234)
FROM 'sysmaster:"informix".sysdual';
```

These are the returned DATETIME values from a server in the United States Pacific time zone:

```
1969-12-31 06:53:53   1970-01-01 01:06:07   2011-03-11 22:56:39
  1902-05-12 01:28:30
  # Server running in TZ=US/Pacific
```

These are the returned DATETIME values from the same query from a server in the UTC0 time zone:

```
1969-12-31 14:53:53   1970-01-01 09:06:07   2011-03-12 06:56:39
  1902-05-12 08:28:30
  # Server running in TZ=UTC0
```

Note that the DAY component in the third DBINFO result is different for the United States Pacific time zone and for the UTC0 time zone, because of the 8-hour offset between those two time zones.

The database server time zone can similarly affect the return value from other expressions for points in time, such as CURRENT, SYSDATE, and TODAY, whose DATETIME YEAR TO SECOND or DATE representation depends on the time zone of the server.

Encryption and decryption functions

supports built-in encryption and decryption functions.

The encryption functions **ENCRYPT_AES** and **ENCRYPT_TDES** return an *encrypted_data* value that encrypts the *data* argument. Conversely, decryption functions **DECRYPT_CHAR** and **DECRYPT_BINARY** return a plain-text *data* value from the *encrypted_data* argument. Use this syntax to call these functions:

Encryption and Decryption Functions

“ { { **ENCRYPT_AES** | **ENCRYPT_TDES** } (*data* [, *password* [, *hint*]]) | { **DECRYPT_CHAR** | **DECRYPT_BINARY** } (*encrypted_data* [, *password*]) | **GETHINT** (*encrypted_data*) } ”

Element	Description	Restrictions	Syntax
data	A plain text character string, variable, or large object of type BLOB or CLOB to be encrypted	Must be a character or BLOB data type	Expression on page 1003
encrypted_data	A character string or variable containing output from **ENCRYPT_AES** or from **ENCRYPT_TDES**	Decryption requires the encryption password	Expression on page 1003
hint	A character string that you define here. Default is the value from the WITH HINT clause of the SET ENCRYPTION statement that defined *password*.	No more than 32 bytes	Quoted String on page 1202
password	A character string that the encryption function defines. Default is the session password value defined by the SET ENCRYPTION statement	At least 6 bytes, but no more than 128 bytes	Quoted String on page 1202

You can invoke these encryption and decryption functions from within DML statements or with the EXECUTE FUNCTION statement.

For distributed operations over a network, all participating database servers must support these (or equivalent) functions. If the network is not secure, the DBSA must enable the *encryption communication support module* (ENCCSM) to provide data encryption between the database server and client systems, in order to avoid transmitting passwords as plain text.

Encryption or decryption calls slow the performance of the SQL statement within which these functions are invoked, but have no effect on other statements. However, if you store encrypted data in a column that is an index key, or in a column on which a constraint is defined, Informix® cannot enforce the constraint, and DML statements cannot use the index.

Similarly, do not encrypt a column whose value is referenced in the fragment key expression of a fragmented table.

You cannot encrypt the security label in a column of type IDSSECURITY label.

Column Level and Cell Level Encryption

The encryption and decryption functions can support two ways of using data encryption features, namely *column level* and *cell level* encryption.

- Column level encryption means that all values in a given column are encrypted with the same password (which can be a word or phrase), the same cipher, and the same cipher mode.

  Users of this form of encryption should consider not using the *hint* feature of these functions, but instead store a mnemonic hint for remembering the password in some other location. Otherwise, the same *hint* will occupy disk space in every row that contains an encrypted value.
- Cell level encryption means that within a column of encrypted data many different passwords (or different ciphers or cipher modes) are used.

This use of encryption is also called *row-column level* or *set-column level* encryption. Compared to column-level encryption, this makes the task of data management more complex, because if different passwords are required for decrypting different rows of the same table, it is not possible to write a single SELECT statement to fetch all the decrypted data. In some situations, however, individual users may need this technique to protect personal data.

To protect data security and confidentiality, the database server does not store information in the system catalog to indicate whether any table (or any column or row) includes encrypted data. Similarly, the logical logs of Informix® do not record SET ENCRYPTION statements, nor calls to encryption or decryption functions. (The Trusted Facility feature for secure auditing, however, can use the 'STEP' audit-event mnemonic to record execution of the SET ENCRYPTION statement, and can use the 'CRPT' audit-event mnemonic to record successful or unsuccessful calls to **DECRYPT_CHAR** or **DECRYPT_BINARY**.)

The Password and Hint Specifications

The SET ENCRYPTION statement or an encryption function can define a password and hint for the current session. The *password* must be specified as a character expression that returns at least 6 bytes, but no more than 128. The optional *hint* is specified as a character expression that returns no more than 32 bytes.

The purpose of the *hint* is to help users to remember the *password*. When you call **ENCRYPT_AES** or **ENCRYPT_TDES** with a *hint* argument, it is encrypted and embedded in the *encrypted_data*, from which **GETHINT** can retrieve it. But if you define *hint* as NULL, or omit *hint* when SET ENCRYPTION specified no default *hint* for the *session password*, no *hint* is embedded in the *encrypted_data*.

The *password* used for encryption and decryption is either the *password* argument to the function, or if you omit this argument, it is the *session password* specified in the last SET ENCRYPTION statement executed before you invoke the function.

The **DECRYPT_CHAR**, **DECRYPT_BINARY**, or **GETHINT** function call fails with an error if the *encrypted_data* argument is not in an encrypted format, or if the *password* argument to a decryption function is omitted when no *session password* value was set by SET ENCRYPTION. An error also results if the *password* used for decryption is not the same *password* used for encryption.

Encryption key management, which is critical to the secure operation of the database, is delegated entirely to the application. This implementation means that the *password* itself is not stored in the database. Without help from the user through the application, the database server cannot decrypt the encrypted data.

If you invoke any of these functions from a UDR, you might prefer to set a *session password* in the SET ENCRYPTION statement. Otherwise, *password* will be visible to users who can view the **sysprocbody.data** column in the system catalog.

Data Types, Encoding, and Size of Encrypted Values

The *data* and corresponding *encrypted_data* arguments can be of any built-in character type (CHAR, LVARCHAR, NCHAR, NVARCHAR, or VARCHAR), or can be a smart large object of type BLOB or CLOB. (Use CLOB in place of TEXT, which these functions do not support.)

Corresponding *data* and *encrypted_data* values that the encryption or decryption functions return have the same character, BLOB, or CLOB data type, except in cases where encryption of a VARCHAR or NVARCHAR string would return an overflow error. For operations on CHAR, LVARCHAR, NCHAR, NVARCHAR, or VARCHAR data and on *encrypted_data* values, the encryption and decryption functions follow the data-type promotion rules of **CONCAT** and the other SQL string manipulation functions for the data type of their return value, as in the following examples.

- If the VARCHAR *data* argument to **ENCRYPT_TDES** (with no hint) is a 200 byte string, then Informix® automatically promotes the returned value to an LVARCHAR data type, because the encrypted value exceeds the 255 byte limit for VARCHAR objects.
- If the NVARCHAR *encrypted_data* argument to **ENCRYPT_AES** (with a hint) is a string 200 bytes long, then Informix® automatically promotes the returned value to an NCHAR data type, because the encrypted value exceeds the 255 byte limit for NVARCHAR objects.

For more information about return type promotion for character strings that the encryption, decryption, and certain other string-manipulation functions return, see Return Types from the CONCAT Function on page 1115. The first table in that section describes *data* or *encrypted_data* arguments that are not smart large objects. (For smart large object arguments, the return type is a BLOB or CLOB object.)

The encryption or decryption function call returns overflow error -881, however, if the return value exceeds the 32,767-byte limit for CHAR, NCHAR strings, or the 32,739-byte limit for LVARCHAR strings. To avoid this error, use BLOB or CLOB objects as the *data* or *encrypted_data* argument, rather than a character data type, when the encryption or decryption operation requires an argument or a return value that might be larger than the (approximately 32Kb) limit for character data types.

Except for original *data* of BLOB or CLOB data types, the *encrypted_data* value is encoded in BASE64 format. An encrypted value requires more space than the corresponding plain text, because the database must also store the information (except for the encryption key) that is needed for decryption. If a *hint* is used, it adds to the length of *encrypted_data*.

The BASE64 encoding scheme stores 6 bits of input data as 8 bits of output. To encode N bytes of data, BASE64 requires at least ((4N+3)/3) bytes of storage, where the slash character (/) represents integer division. Padding and headers can increase BASE64 storage requirements above this ((4N+3)/3) ratio. Example of Column Level Encryption on page 1083 lists formulae to estimate the size of data values encrypted in BASE64 format. It typically requires changes to the schema of an existing table that will store BASE64 format encrypted data, especially if a hint will also be stored.

The following table shows how the data type of the input string corresponds to the data type of the value that **ENCRYPT_AES** or **ENCRYPT_TDES** returns:

Table 25. Data Types for ENCRYPT_AES and ENCRYPT_TDES Functions

Plain Text Data Type	Encrypted Data Type	Decryption Function
CHAR	CHAR	**DECRYPT_CHAR**
NCHAR	NCHAR	**DECRYPT_CHAR**
VARCHAR	VARCHAR or CHAR	**DECRYPT_CHAR**
NVARCHAR	NVARCHAR or NCHAR	**DECRYPT_CHAR**

Table 25. Data Types for ENCRYPT_AES and ENCRYPT_TDES Functions (continued)

Plain Text Data Type	Encrypted Data Type	Decryption Function
LVARCHAR	LVARCHAR	**DECRYPT_CHAR**
BLOB	BLOB	**DECRYPT_BINARY**
CLOB	BLOB	**DECRYPT_CHAR**

Columns of type VARCHAR and NVARCHAR store no more than 255 bytes. If the *data* string is too long for these data types to store both the encrypted data and encryption overhead, then the value returned by the encryption function is automatically changed from VARCHAR or NVARCHAR into a fixed CHAR or NCHAR value, with no trailing blanks in the encoded encrypted value.

Encrypted values of type BLOB or CLOB are not in BASE64 encoding format, and their size increase after encryption is independent of the original data size. For BLOB or CLOB values, the encrypted size (in bytes) has the following formula, where N is the original size of the plain text, and H is the size of the unencrypted hint string, if encryption is performed by **ENCRYPT_TDES**:

```
N + H + 24 bytes.
```

For BLOB or CLOB values that **ENCRYPT_AES** encrypts, the overhead is larger:

```
N + H + 32 bytes.
```

Example of Column Level Encryption

The following example illustrates how to use the built-in encryption and decryption functions of Informix® to create and use a table that stores encrypted credit card numbers in a column that has a character data type.

For purposes of this example, assume that the plain text of the values to be encrypted consists of strings of 16 digits. Because encryption functions support character data types, these values are stored in a CHAR column rather than in an INT, BIGINT, or INT8 column.

Calculating storage requirements for encrypted data

The **LENGTH** function provides a convenient way to calculate the storage requirements of encrypted data directly:

```
EXECUTE FUNCTION LENGTH(ENCRYPT_TDES("1234567890123456", "simple password"));
```

This returns `55`.

```
EXECUTE FUNCTION LENGTH(ENCRYPT_TDES("1234567890123456", "simple password",
"12345678901234567890123456789012"));
```

This returns `107`.

```
EXECUTE FUNCTION LENGTH(ENCRYPT_AES("1234567890123456", "simple password"));
```

This returns `67`.

```
EXECUTE FUNCTION LENGTH(ENCRYPT_AES("1234567890123456", "simple password",
"12345678901234567890123456789012"));
```

This returns `119`.

The required storage size for encrypted data is sensitive to three factors:

- N, the number of bytes in the plain text
- whether or not a hint is provided
- which encryption function you use (**ENCRYPT_TDES** or **ENCRYPT_TDES**)

The following formulae describe the four possible cases, and are not simplified:

- Encryption by **ENCRYPT_TDES( )** with no hint:

  ```
  Encrypted size = (4 x ((8 x((N + 8)/8) + 10)/3) + 11)
  ```

- Encryption by **ENCRYPT_AES( )** with no hint:

  ```
  Encrypted size = (4 x ((16 x((N + 16)/16) + 10)/3) + 11)
  ```

- Encryption by **ENCRYPT_TDES( )** with a hint:

  ```
  Encrypted size = (4 x ((8 x((N + 8)/8) + 50)/3) + 11)
  ```

- Encryption by **ENCRYPT_AES( )** with a hint:

  ```
  Encrypted size = (4 x ((16 x((N + 16)/16) + 50)/3) + 11)
  ```

The integer division (/) returns an integer quotient and discards any remainder.

Based on these formulae, the following table shows the encrypted size (in bytes) for selected ranges of values of N:

N	ENCRYPT_TDES No Hint	ENCRYPT_AES No Hint	ENCRYPT_TDES With Hint	ENCRYPT _AES With Hint
1 to 7	35	43	87	99
8 to 15	43	43	99	99
16 to 23	55	67	107	119
24 to 31	67	67	119	119
32 to 39	75	87	131	139
40 to 47	87	87	139	139
100	163	171	215	227
200	299	299	355	355
500	695	707	747	759

If the column size is smaller than the data size returned by encryption functions, the encrypted value is truncated when it is inserted. In this case, it will not be possible to decrypt the data, because the header will indicate that the length should be longer than the data value that the column contains.

These formulae and the values returned by the **LENGTH** function, however, indicate that the table schema in the next example can store the encrypted form of 16-digit credit card numbers (with a hint).

Implementing column-level encryption

The following steps create a table from which a user who knows the password can retrieve rows that include one column of encrypted data.

1. Create a database table containing at least one column of type BLOB, CLOB, or a character data type of sufficient length to store the encrypted values. For example, the following statement creates a table called **customer** in which the column **creditcard** can store encrypted credit card numbers:

   ```
   CREATE TABLE customer (id CHAR(20), creditcard CHAR(107));
   ```

2. Specify a password (and optional hint) and insert encrypted data:

   ```
   SET ENCRYPTION PASSWORD 'credit card number is encrypted'
      WITH HINT 'Why is this difficult to read?';
   INSERT INTO customer VALUES ('Alice',
      encrypt_tdes('1234567890123456'));
   INSERT INTO customer VALUES ('Bob',
      encrypt_tdes('2345678901234567'));
   ```

3. Query the encrypted data, using a decryption function:

   ```
   SELECT id, DECRYPT_CHAR(creditcard,
      'credit card number is encrypted') FROM customer;
   ```

   The following query calls a decryption function in the WHERE clause, using the *session password* default, rather than an explicit *password* argument:

   ```
   SELECT id FROM customer
      WHERE DECRYPT_CHAR(creditcard) = '2345678901234567';
   ```

Column level encryption offers the coding convenience of passing the implicit *session password* for all rows with encrypted columns, and in multiple encryption and decryption function calls in the same SQL statement. Confidentiality of the data, however, requires users who know the password on encrypted columns to avoid compromising its secrecy. Triggers and UDRs, for example, should always use the *session password*, rather than explicit *password* arguments if they invoke the encryption or decryption functions.

The DBSA can manage highly confidential data with column level encryption. Informix® does not, however, prevent users with sufficient privileges from entering data encrypted by some other password into a table whose other rows use the designated column level encryption password.

DECRYPT_CHAR Function

The **DECRYPT_CHAR** function accepts as its first argument an *encrypted_data* character string that can have any character type (CHAR, LVARCHAR, NCHAR, NVARCHAR, or VARCHAR). You must specify a *password* as its second argument, unless the SET ENCRYPTION statement has specified for this session the same *session password* by which the first argument was encrypted.

The **DECRYPT_CHAR** function also accepts as its first argument an *encrypted_data* large object of type BLOB or CLOB. You must specify a password as its second argument, unless the SET ENCRYPTION statement has specified as the default for this session the same *password* by which the first argument was encrypted. If the call to **DECRYPT_CHAR** is successful, it returns a CLOB large object that contains the plain text version of the *encrypted_data* argument.

If the call to **DECRYPT_CHAR** with an encrypted string argument is successful, it returns a character string that contains the plain text version of the *encrypted_data* argument. The following example returns a character string containing a decrypted value from the **ssid** column of the **engineers** table for the row whose **empno** value is 287:

```
SELECT DECRYPT_CHAR (ssid) FROM engineers WHERE empno = 287;
```

If the first argument to **DECRYPT_CHAR** is not an encrypted value, or if the second argument (or the default *password* specified by SET ENCRYPTION) is not the *password* that was used when the first argument was encrypted, Informix® issues an error, and the call to **DECRYPT_CHAR** fails. (See the description of the GETHINT Function on page 1088 for one possible action to take when you cannot remember the *password* that was used for encryption.)

Do not use **DECRYPT_CHAR** (or any other decryption function) to create a functional index on an encrypted column. This would store the decrypted values as plain text data in the database, defeating the purpose of encryption.

For additional information about using data encryption in column values of Informix® databases, see Encryption and decryption functions on page 1079, and #unique_1626.

DECRYPT_BINARY Function

The **DECRYPT_BINARY** function accepts as its first argument an *encrypted_data* large object of type BLOB or CLOB. You must specify a *password* as its second argument, unless the SET ENCRYPTION statement has specified as the default for this session the same *password* by which the first argument was encrypted.

If the call to **DECRYPT_BINARY** is successful, it returns a BLOB or CLOB large object that contains the plain text version of the *encrypted_data* argument. The decrypted BLOB or CLOB object is temporarily stored in the default sbspace that the SBSPACENAME configuration parameter setting specifies.

If the first argument to **DECRYPT_BINARY** is an encrypted value of a character data type, Informix® invokes the **DECRYPT_CHAR** function and attempts to decrypt the specified value.

If the first argument to **DECRYPT_BINARY** is not an encrypted value, or if the second argument (or the default *password* specified by SET ENCRYPTION) is not the *password* that was used when the first argument was encrypted, Informix® issues an error, and the call to **DECRYPT_BINARY** fails. (See the description of the GETHINT Function on page 1088 for one possible action to take when you cannot remember the *password* that was used for encryption.)

Do not use **DECRYPT_BINARY** (or any other decryption function) to create a functional index on an encrypted column. This would store the decrypted values as plain text data in the database, defeating the purpose of encryption.

For additional information about using data encryption in column values of Informix® databases, see Encryption and decryption functions on page 1079, and #unique_1626.

ENCRYPT_TDES Function

The **ENCRYPT_TDES** function returns a value that is the result of encrypting a character expression, or a BLOB or CLOB value, by applying the TDES (Triple Data Encryption Standard, which is sometimes also called DES3) algorithm to its first argument. This algorithm is slower than the AES algorithm that is used by the **ENCRYPT_AES** function, but is considered somewhat more secure. The disk space required as encryption overhead resembles that of **ENCRYPT_AES**, but is somewhat smaller because of the smaller block size of ENCRYPT_TDES. (See "Calculating storage requirements for encrypted data" Calculating storage requirements for encrypted data on page 1083 for a discussion of how to estimate the size of encrypted character strings.) For BLOB or CLOB values, the encrypted object is temporarily stored in the default sbspace that the SBSPACENAME configuration parameter specifies.

Those differences in performance, tamper-resistance, and in the returned *encrypted_data* size that the previous paragraph lists are the practical differences between the **ENCRYPT_TDES** and **ENCRYPT_AES** functions, which otherwise follow the same rules, defaults, and restrictions that appear in the description of **ENCRYPT_AES** on the previous page in regard to the following features:

- The required first argument (the plain text *data* value to be encrypted)
- The explicit or default second argument (the *password* string that must also be an argument to **DECRYPT_CHAR** or **DECRYPT_BINARY** to decrypt the returned *encrypted_data* value). This must be specified unless a default *session password* has been set by the SET ENCRYPTION statement
- The optional third argument (the *hint* value) that might assist users who forget the *password*. If you subsequently cannot remember an explicit or default *hint* that was defined for *password*, you can use the returned value from **ENCRYPT_TDES** as the argument to **GETHINT** to retrieve the *hint*.

The following example calls **ENCRYPT_TDES** from the SET clause of an UPDATE statement. Here the *session password* is 'PERSEPHONE' and the *hint* string is "pomegranate", with column **colU** of table **tabU** the *data* argument. Because the WHERE clause condition of "1=1" is true for all rows of **tabU**, the effect of this statement is to replace every plain text **colU** value with encrypted strings returned by the algorithm that **ENCRYPT_TDES** implements:

```
EXEC SQL SET ENCRYPTION PASSWORD 'PERSEPHONE' WITH HINT 'pomegranate';
EXEC SQL UPDATE tabU SET colU = ENCRYPT_TDES (colU) WHERE 1=1;
```

This example assumes that the character data type of **colU** is of sufficient size to store the new encrypted values without truncation. (A more cautious example might execute an appropriate ALTER TABLE statement before the UPDATE.)

For additional information about using data encryption in column values of Informix® databases, see Encryption and decryption functions on page 1079, and #unique_1626.

GETHINT Function

The **GETHINT** function returns a character string that a previously executed SET ENCRYPTION PASSWORD statement defined for the *password* that was used when *encrypted_data* was encrypted by the **ENCRYPT_AES** function or by the **ENCRYPT_TDES** function. This *hint* string typically provides information that helps the user to specify the *password* needed to return the plain text version of *encrypted_data* with the **DECRYPT_CHAR** or **DECRYPT_BINARY** decryption function. The *hint* string, however, should not be the same as the *password*. In the following example, a query returns the *hint* string into a host variable called **myhint**:

```
EXEC SQL SELECT GETHINT(creditcard) INTO :myhint
   FROM customer WHERE id = :myid;
```

An error is returned, rather than a *hint* string, if the *encrypted_data* argument to the **GETHINT** function is not an encrypted string or an encrypted large object.

For additional information about using data encryption in column values of Informix® databases, see Encryption and decryption functions on page 1079, and #unique_1626.

Exponential and Logarithmic Functions

Exponential and logarithmic functions take at least one argument and return a FLOAT data type.

The exponential and logarithmic functions have the following syntax.

Exponential and Logarithmic Functions

" { **EXP**(*float_expression*) | **LN**(*float_expression*) | **LOGN**(*float_expression*) | **LOG10** (*float_expression*) } "

Element	Description	Restrictions	Syntax
float_expression	An argument to the **EXP**, **LN**, **LOGN**, or **LOG10** functions. For the meaning of *float_expression* in these functions, see the individual heading for each function on the pages that follow.	The domain is the set of real numbers, and the range is the set of positive real numbers	Expression on page 1003

EXP Function

The **EXP** function returns the exponent of a numeric expression.

The following example returns the exponent of 3 for each row of the **angles** table:

```
SELECT EXP(3) FROM angles;
```

For this function, the base is always *e*, the base of natural logarithms, as the following example shows:

```
e=exp(1)=2.718281828459
```

When you want to use the base of natural logarithms as the base value, use the **EXP** function. If you want to specify a particular value to raise to a specific power, see the POW Function on page 1060.

LN function

The **LN** function is an alias for the **LOGN** function, and returns the natural logarithm of a numeric argument. This value is the inverse of the exponential value.

LN Function

" `LN(float_expression )` "

The following query returns the natural logarithm of **population** for each row in the **history** table:

```
SELECT LN(population) FROM history WHERE country='US'
   ORDER BY date;
```

LOG10 Function

The **LOG10** function returns the log of a value to base 10. The following example returns the log base 10 of distance for each row of the **travel** table:

```
SELECT LOG10(distance) + 1 digits FROM travel;
```

LOGN Function

The **LOGN** function returns the natural logarithm of a numeric argument.

This return value is the inverse of the exponential value that the **EXP** function returns from the same argument.

The following query returns the natural log of **population** for each row of the **history** table:

```
SELECT LOGN(population) FROM history WHERE country='US'
   ORDER BY date;
```

NVL2 Function

Returns the second argument when the first argument is not NULL. If the first argument is NULL, the third argument is returned.

NVL2 Function

" `NVL2(expression,result-expression,else-expression )` "

The NVL2 function is a synonym for the following code:

```
CASE WHEN expression IS NOT NULL
     THEN result-expression
     ELSE else-expression
```

HEX Function

The **HEX** function returns the hexadecimal encoding of an integer expression.

HEX Function

"{ **HEX**(*int_expression*) }"

Element	Description	Restrictions	Syntax
int_expression	Expression for which you want the hexadecimal equivalent	Must be a literal integer or some other expression that returns an integer	Expression on page 1003

The next example displays the data type and column length of the columns of the **orders** table in hexadecimal format. For MONEY and DECIMAL columns, you can then determine the precision and scale from the lowest and next-to-the-lowest bytes. For VARCHAR and NVARCHAR columns, you can determine the minimum space and maximum space from the lowest and next-to-the-lowest bytes. For more information about encoded information, see the *HCL® Informix® Guide to SQL: Reference*.

```
SELECT colname, coltype, HEX(collength)
   FROM syscolumns C, systables T
   WHERE C.tabid = T.tabid AND T.tabname = 'orders';
```

The following example lists the names of all the tables in the current database and their corresponding tblspace number in hexadecimal format.

```
SELECT tabname, HEX(partnum) FROM systables;
```

The two most significant bytes in the hexadecimal number constitute the dbspace number. They identify the table in **oncheck** output in Informix®.

The **HEX** function can operate on an expression, as the next example shows:

```
SELECT HEX(order_num + 1) FROM orders;
```

Length functions

Use length functions to determine the length of a character column, string, or variable, or of the value returned by a character expression, or (for **CHAR_LENGTH** in multibyte locales) the number of logical characters.

(explicit id unique_1461_Connect_42_lefu001) unique_1461_Connect_42_lefu001(explicit id unique_1461_Connect_42_lefu002)

unique_1461_Connect_42_lefu002(explicit id unique_1461_Connect_42_lefu003) unique_1461_Connect_42_lefu003(explicit id

unique_1461_Connect_42_lefu004) unique_1461_Connect_42_lefu004

Length Functions

" [] "

" { { LENGTH | LEN } | { CHAR_LENGTH | CHARACTER_LENGTH } | OCTET_LENGTH } "

" ({ <Quoted String>[] | [][] variable_name | [table.] column }) "

Element	Description	Restrictions	Syntax
column	Name of a column in *table*	Must have a character data type	Identifier on page 1231
table	Name of the table in which the specified column occurs	Must exist	Identifier on page 1231
variable	Host variable or SPL variable that contains a character string	Must have a character data type	See language-specific rules for names.

Each of these functions has a distinct purpose:

- **LENGTH** (also known as **LEN**)
- **OCTET_LENGTH**
- **CHAR_LENGTH** (also known as **CHARACTER_LENGTH**)

LENGTH Function

The **LENGTH** function (also called **LEN**) returns the number of bytes in a character column, but excluding any trailing blank spaces.

For BYTE or TEXT columns, **LENGTH** returns the full number of bytes, including any trailing blank spaces.

In , **LENGTH** can also return the length of a character variable.

The following example illustrates the use of the **LENGTH** function:

```
SELECT customer_num, LENGTH(fname) + LENGTH(lname),
   LENGTH('How many bytes is this?')
   FROM customer WHERE LENGTH(company) > 10;
```

The next example calls the function by its other name, **LEN**;

```
EXECUTE FUNCTION LEN("www.ibm.com");
```

The SQL statement above returns the integer value `11`.

See also the discussion of **LENGTH** in the *HCL® Informix® GLS User's Guide*.

OCTET_LENGTH Function

The **OCTET_LENGTH** returns the number of bytes in a character column, including any trailing blank spaces. See also the *HCL® Informix® GLS User's Guide*.

CHAR_LENGTH Function

The **CHAR_LENGTH** function returns the number of logical characters in its argument, which can be a character column, a character variable, or a quoted string. This built-in function can also be invoked as **CHARACTER_LENGTH**.

In the default U.S. English locale and other single-byte locales, **CHAR_LENGTH** behaves exactly like the **LENGTH** function, and returns the number of bytes in its argument.

For multibyte code sets, however, which various Unicode, East Asian, and other nondefault locales support, the return value can be less than the number of bytes in the argument. For a discussion of this function, see the *HCL® Informix® GLS User's Guide*.

Security Label Support Functions

The security label support functions enable users to manipulate security labels. A security label can be referenced in three different ways:

- A name, as declared in the CREATE SECURITY LABEL or RENAME SECURITY LABEL statement.
- A list of values for each component of the security policy of the security label.
- An internal encoded value that the IDSSECURITYLABEL data type stores.

These functions can convert between the various forms of a security label. They are typically used to specify a label in DML operations on data rows that are secured by label-based access control (LBAC). In these operations, however, the security label support functions do not provide any more access to protected data than is already provided by the security credentials of the user who invokes the function.

Security Label Support Function

" { **SECLABEL_TO_CHAR**('*policy*',*column*) | **SECLABEL_BY_COMP**('*policy*','{ *component* | (*element*) } ') |
SECLABEL_BY_NAME ('*policy*', '*label*') } "

Element	Description	Restrictions	Syntax
column	A column of type IDSSECURITYLABEL	Must exist and must store a label of the *policy*	Identifier on page 1231
component	Value of a component of the *policy*	Must exist and must be a component of the *policy*	Quoted String on page 1202

Element	Description	Restrictions	Syntax
element	Value of an element within a list of values of the *component*	Must exist and must be elements of a single component of the *policy*	Quoted String on page 1202
label	Identifier of the security label whose value the function returns	Must exist and must be a label of the *policy*	Quoted String on page 1202
policy	The security policy supported by the security label whose value the function returns	Must exist and must be the security policy that secures the table	Quoted String on page 1202

These functions return a security label of the specified security *policy*. They can be used within DML statements that reference a protected database table, but they can also evaluate to a security label in other calling contexts. Each of these functions requires a different argument list:

- **SECLABEL_TO_CHAR** requires the security policy name and an expression that returns a IDSSECURITYLABEL object, such as the name of a column of that data type.
- **SECLABEL_BY_COMP** requires the security policy name and the values of the individual components of the security label.
- **SECLABEL_BY_NAME** requires the names of the security policy and of the security label.

SECLABEL_BY_NAME Function

The **SECLABEL_BY_NAME** function enables users to provide a security label directly by specifying its name.

The following INSERT statement inserts a row into table **T1**, which is protected by the security policy called '**MegaCorp**'. The VALUES clause of the INSERT statement provides the security label '**mylabel**' for the row to be inserted by using the **SECLABEL_BY_NAME** function.

```
INSERT INTO T1 VALUES (SECLABEL_BY_NAME ('MegaCorp', 'mylabel'), 1, 'xyz');
```

The success of this **SECLABEL_BY_NAME** function call does not guarantee success of the INSERT operation in this example, because whether or not the user has sufficient security credentials to insert the label **mylabel** into the row is subject to the **IDSLBACWRITE** rules of the **MegaCorp** security policy.

SECLABEL_BY_COMP Function

The **SECLABEL_BY_COMP** function returns an IDSSECURITYLABEL object, which is a security label in its internal encoded string format. This function enables users to provide a security label directly by specifying its component values.

If a security label component requires multiple values, then such multiple values can be specified by putting those values between parenthesis as in `(value_1, value_2, ...)`. When a component in a particular security label needs to be empty, it can be specified by putting nothing between an opening and a closing parenthesis, as in `()`. Because the blank space (ASCII 32) is a valid character in an element value for a security component, any blank space appearing in the security label string is treated as part of the element value for that component.

The security label string is limited to a maximum of 32 kilobytes. An error is returned if the string length exceeds this limit.

The following INSERT statement inserts a row into table **T1** which is protected by the security policy called '**MegaCorp**' that has three components: '**level**', '**compartments**', and '**groups**'. Here the user provides a security label for the row to be inserted by specifying the **SECLABEL_BY_COMP** function. The security label in this example has the value '**VP**' for the **level** component, the value '**Marketing**' for the **compartments** component, and the value '**West**' for the **groups** component. In the arguments to **SECLABEL_BY_COMP**. colon symbols separate these security component element values, and quotation marks delimit the list of component values of the security label.

```
INSERT INTO T1
   VALUES (SECLABEL_BY_COMP ('MegaCorp', 'VP:Marketing:West'), 1, 'xyz';
```

In the next example, the INSERT statement inserts a row in table **T1** which is protected by the same **MegaCorp** security policy, which has the same three components as in the previous example: **level**, **compartments**, and **groups**. The user provides the security label for the row to be inserted by specifying the policy name and a list of security component elements as arguments to the **SECLABEL_BY_COMP** function. Here the security label has the value '**Director**' for the **level** component, the values '**HR**' and '**Finance**' for the **compartments** component, and the value '**East**' for the **groups** component.

```
INSERT INTO T1
   VALUES (SECLABEL_BY_COMP ('MegaCorp', 'Director:(HR,Finance):East'), 1, 'xyz');
```

The following example inserts a row into table **T1** which is protected by the **MegaCorp** security policy, whose three components are **level**, **compartments**, and **groups**. The **SECLABEL_BY_COMP** function specifies the security label for the row to be inserted. The security label in this example has the value '**CEO**' for level **component**, the empty set for the **compartments** component, and the value '**EntireRegion**' for the **groups** component.

```
INSERT INTO T1
   VALUES (SECLABEL_BY_COMP ('MegaCorp', 'CEO:():EntireRegion'), 3, 'abc');
```

As in all of these examples, the success of the **SECLABEL_BY_COMP** function call does not guarantee the success of the INSERT statement, because the security credentials of the user are first compared to the security label that protects table **T1**, using the IDSLBACRWRITE rules of the **MegaCorp** security policy, before the database server allows or denies write access for inserting the new row.

SECLABEL_TO_CHAR Function

The **SECLABEL_TO_CHAR** function returns a security label in the security label string format.

The security credentials of the user executing this function can affect the output of the function. An element of a security label component is not included in the output if the user does not have read access to that element. A user has read access to an element if the security credentials of the user provide read access to data that is protected by a security label containing only that element and no other elements.

For the rule set **IDSLBACRULES**, only components of type TREE can contain elements to which a user does not have read access to a subset of elements. For other types of component, if any element blocks read access, then the user cannot read the row at all. Thus, only security components of type TREE can have a subset of security component elements excluded in this way.

For example, if the TREE type component of the security label of a user is {**A**} and the TREE type component of a row security label is {**A**, **B**}, then only component **A** is returned, and the user is not aware that **B** existed in the row security label. If the user holds an exemption on the **IDSLBACREADTREE** rule, however, the returned security components are both **A** and **B**.

In the next example, the **MegaCorp** security policy has a security label called **mylabel** that consists of a **level** component whose value is '**Director**', and a **compartments** component with the values '**HR**' and '**Finance**.' A user to whom '**mylabel**' was granted has inserted a row with that security label into table **T1**. In this context, the security label string returned by the **SECLABEL_TO_CHAR** function in the following SELECT statement on **T1** is as follows.

```
SELECT SECLABEL_TO_CHAR ('MegaCorp', C1) FROM T1;

Row returned:

'Director:(HR,Finance)'
```

The success of this query implies that the **SECLABEL_TO_CHAR** function succeeded, and that the security credentials of the user were sufficient, according to the **IDSLBACREAD** rules of the **MegaCorp** security policy, for the database server to allow read access to the values of the security policy name and of the security label components."

The security label string is limited to a maximum size of 32 kilobytes. If the length of the security label string to be returned exceeds this upper limit, a warning is issued, and a truncated 32 kilobyte string is returned.

SIGN function

The SIGN function returns an indicator of the sign of the argument.

SIGN Function

" **SIGN**(*expression*) "

If the argument is less than zero, `-1` is returned. If the argument equals zero, `0` is returned. If the argument is greater than zero, `1` is returned. The result returned is always an integer with one of these values.

Smart-Large-Object Functions

The smart-large-object functions support objects of BLOB and CLOB data types:

The smart-large-object functions have the following syntax:

Smart-Large-Object Functions

" { { **FILETOBLOB** | **FILETOCLOB** } (*'pathname'*, *'file_destination'* [, *'table'* , *'column'*]) | **LOTOFILE**({ *BLOB_column* | *CLOB_column* } , *'pathname'*, *'file_destination'*) | **LOCOPY**({ *BLOB_column* | *CLOB_column* } [, *'table'* , *'column'*]) } "

Element	Description	Restrictions	Syntax
BLOB_*column*, CLOB_*column*	A column of type BLOB; a column of type CLOB	The *column* data type must be BLOB or CLOB	Identifier on page 1231
column	Column within *table* for the copy of the BLOB or CLOB value	Must have CLOB or BLOB as its data type	Quoted String on page 1202
file_destination	The system on which to put or get the smart large object	The only valid values are the strings `'server'` or `'client'`	Quoted String on page 1202
pathname	Directory path and filename to locate the smart large object	No more than 256 bytes. Must exist on *file_destination* system. See also Pathnames with Commas on page 1097.	Quoted String on page 1202
table	Table containing *column* for the copy of the BLOB or CLOB value	A comma (not a period) separates the '*table*' and '*column*' arguments	Quoted String on page 1202

FILETOBLOB and FILETOCLOB Functions

The **FILETOBLOB** function creates a BLOB value for data that is stored in a specified operating-system file. Similarly, the **FILETOCLOB** function creates a CLOB value for a data value that is stored in an operating-system file.

These functions determine the operating-system file to use from the following parameters:

- The *pathname* parameter identifies the directory path and name of the source file.
- The *file destination* parameter identifies the computer, `'client'` or `'server'`, on which this file resides:
  - Set *file destination* to `'client'` to identify the client computer as the location of the source file. The *pathname* can be either a full pathname or relative to the current directory.
  - Set *file destination* to `'server'` to identify the server computer as the location of the source file. The *pathname* must be a full pathname.

The *table* and *column* parameters are optional:

- If you omit *table* and *column*, the **FILETOBLOB** function creates a BLOB value with the system-specified storage defaults, and the **FILETOCLOB** function creates a CLOB value with the system-specified storage defaults.

  These functions obtain the system-specific storage characteristics from either the ONCONFIG file or the sbspace. For more information on system-specified storage defaults, see the *HCL® Informix® Administrator's Guide*.
- If you specify *table* and *column*, the **FILETOBLOB** and **FILETOCLOB** functions use the storage characteristics from the specified column for the BLOB or CLOB value that they create.

The **FILETOBLOB** function returns a handle value (a pointer) to the new BLOB value. Similarly, **FILETOCLOB** returns a handle value to the new CLOB value. Neither function actually copies the smart-large-object value into a database column. You must assign the BLOB or CLOB value to the appropriate column.

The **FILETOCLOB** function performs any code-set conversion that might be required when it copies the file from the client or server computer to the database.

The following INSERT statement uses the **FILETOCLOB** function to create a CLOB value from the value in the **smith.rsm** file:

```
INSERT INTO candidate (cand_num, cand_lname, resume)
   VALUES (2, 'Smith', FILETOCLOB('smith.rsm', 'client'));
```

In the preceding example, the **FILETOCLOB** function reads the **smith.rsm** file in the current directory on the client computer and returns a handle value to a CLOB value that contains the data in this file. Because the **FILETOCLOB** function does not specify a table and column name, this new CLOB value has the system-specified storage characteristics. The INSERT statement then assigns this CLOB value to the **resume** column in the **candidate** table.

The following INSERT statement uses the **FILETOBLOB** function to create a BLOB value from the value in the **photos.xxx** file on the local database server, and insert that value into the **election2008** table of the **rdb** database, which is another database of the local database server:

```
INSERT INTO rdb@:election2008 (cand_pic)
   VALUES (FILETOBLOB('C:\tmp\photos.xxx', 'server',
      'candidate', 'cand_photo'));
```

In the preceding example, the **FILETOBLOB** function reads the **photos.xxx** file in the specified directory on the local database server and returns a handle value to a BLOB value that contains the data in this file. The INSERT statement then assigns this BLOB value to the **cand_pic** column in the **election2008** table in the **rdb** database of the local database server. This new BLOB value has the storage characteristics of the **cand_photo** column in the **candidate** table in the local database.

In the following example, the new BLOB value has the storage characteristics of the **cand_pix** column in the **election96** table in the **rdb2** database, where **rdb1** and **rdb2** are databases of the local Informix® instance:

```
INSERT INTO rdb1:election2008 (cand_pic)
   VALUES (FILETOBLOB('C:\tmp\photos.xxx', 'server',
      'rdb2:election96', 'cand_pix'));
```

When you qualify the **FILETOBLOB** or **FILETOCLOB** function with the name of a remote database and a remote database server, the *pathname* and the *file destination* become relative to the remote database server.

When you specify `server` as the file destination, as the following example shows, the **FILETOBLOB** function looks for the source file (in this case, **photos.xxx**) on the remote database server:

```
INSERT INTO rdb@rserv:election (cand_pic)
   VALUES (rdb@rserv:FILETOBLOB('C:\tmp\photos.xxx', 'server'));
```

When you specify `client` as the file destination, however, as in the following example, the **FILETOBLOB** function looks for the source file (in this case, **photos.xxx**) on the local client computer:

```
INSERT INTO rdb@rserv:election (cand_pic)
   VALUES (rdb@rserv:FILETOBLOB('photos.xxx', 'client'));
```

Pathnames with Commas

If a comma (**,**) symbol is within the *pathname* of the function, the database server expects the pathname to have the following format:

```
"offset, length, pathname"
```

For pathnames that contain a comma, you must also specify an offset and length, as in the following example:

```
FILETOBLOB("0,-1,/tmp/blob,x","server");
```

The first term in the quoted *pathname* string is an *offset* of `0`, which instructs the database server to begin reading at the start of the file.

The second term is a *length* of `-1`, which instructs the database server to continue reading until the end of the entire file.

The third term is the */tmp/blob,x pathname*, specifying which file to read. (Notice the comma symbol that precedes the *x*.)

Because the *pathname* includes a comma, the comma-separated *offset* and *length* specifications are necessary in this example to avoid an error when **FILETOBLOB** is called. You do not need to specify *offset* and *length* for pathnames that include no comma, but including 0,-1, as the initial characters of the pathname string avoids this error for any valid pathname.

LOTOFILE Function

The **LOTOFILE** function copies a smart large object to an operating-system file.

The first parameter specifies the BLOB or CLOB column to copy. The function determines what file to create from the following parameters:

- The *pathname* identifies the directory path and the source file name.
- The *file destination* identifies the computer, `'client'` or `'server'`, on which this file resides:
  - Set *file destination* to `'client'` to identify the client computer as the location of the source file. The *pathname* can be either a full pathname or a path relative to the current directory.
  - Set *file destination* to `'server'` to identify the server computer as the location of the source file. The full pathname is required.

By default, the **LOTOFILE** function generates a filename of the form:

```
file.hex_id
```

In this format, *file* is the filename you specify in *pathname* and *hex_id* is the unique hexadecimal smart-large-object identifier. The maximum number of digits for a smart-large-object identifier is 17. Most smart large objects, however, would have an identifier with fewer digits.

For example, suppose that you specify a UNIX™ *pathname* value as follows:

```
'/tmp/resume'
```

If the CLOB column has the identifier **203b2**, then **LOTOFILE** creates the file:

```
/tmp/resume.203b2
```

For another example, suppose that you specify a Windows™ *pathname* value as follows:

```
'C:\tmp\resume'
```

If the CLOB column has an identifier of **203b2**, the **LOTOFILE** function would create the file:

```
C:\tmp\resume.203b2
```

To change the default filename, you can specify the following wildcards in the filename of the *pathname*:

- One or more contiguous question mark (?) characters in the filename can generate a unique filename.

  The **LOTOFILE** function replaces each question mark with a hexadecimal digit from the identifier of the BLOB or CLOB column.

  For example, suppose that you specify a UNIX™ *pathname* value as follows:

  ```
  '/tmp/resume??.txt'
  ```

  The **LOTOFILE** function puts 2 digits of the hexadecimal identifier into the name. If the CLOB column has an identifier of **203b2**, the **LOTOFILE** function would create the file:

  ```
  /tmp/resume20.txt
  ```

  If you specify more than 17 question marks, **LOTOFILE** ignores them.

- An exclamation (!) point at the end of the filename indicates that the filename does not need to be unique.

  For example, suppose that you specify a Windows™ pathname value as follows:

  ```
  'C:\tmp\resume.txt!'
  ```

  The **LOTOFILE** function does not use the smart-large-object identifier in the filename, so it generates the following file:

  ```
  C:\tmp\resume.txt
  ```

If the filename that you specify already exists, **LOTOFILE** returns an error.

The **LOTOFILE** function performs any code-set conversion that might be required when it copies a CLOB value from the database to a file on the client or server computer.

When you qualify **LOTOFILE** with the name of a remote database and a remote database server, the BLOB or CLOB column, the *pathname*, and the *file destination* become relative to the remote database server.

When you specify `server` as the file destination, as in the next example, the **LOTOFILE** function copies the smart large object from the remote database server to a source file in the specified directory on the remote database server:

```
rdb@rserv:LOTOFILE(blob_col, 'C:\tmp\photo.gif!', 'server')
```

If you specify `client` as the file destination, as in the following example, the **LOTOFILE** function copies the smart large object from the remote database server to a source file in the specified directory on the local client computer:

```
rdb@rserv:LOTOFILE(clob_col, 'C:\tmp\essay.txt!', 'client')
```

LOCOPY Function

The **LOCOPY** function creates a copy of a smart large object.

The first parameter specifies the BLOB or CLOB column to copy. The *table* and *column* parameters are optional.

- If you omit *table* and *column* arguments, the **LOCOPY** function creates a smart large object with system-specified storage defaults, and copies the data in the BLOB or CLOB column into it.

  The **LOCOPY** function obtains the system-specific storage defaults from either the ONCONFIG file or the sbspace. For more information on system-specified storage defaults, see the *HCL® Informix® Administrator's Guide*.
- When you specify *table* and *column*, the **LOCOPY** function uses the storage characteristics from the specified *column* for the BLOB or CLOB value that it creates.

The **LOCOPY** function returns a handle value (a pointer) to the new BLOB or CLOB value. This function does *not* actually store the new smart-large-object value into a column in the database. You must assign the BLOB or CLOB value to the appropriate column.

The following code fragment copies the CLOB value in the **resume** column of the **candidate** table to the **resume** column of the **interview** table:

```
/* Insert a new row in the interviews table and get the
 * resulting SERIAL value (from sqlca.sqlerrd[1])
 */
EXEC SQL insert into interviews (intrv_num, intrv_time)
   values (0, '09:30');
intrv_num = sqlca.sqlerrd[1];

/* Update this interviews row with the candidate number
 * and resume from the candidate table. Use LOCOPY to
 * create a copy of the CLOB value in the resume column
 * of the candidate table.
 */
EXEC SQL update interviews
   SET (cand_num, resume) =
      (SELECT cand_num,
         LOCOPY(resume, 'candidate', 'resume')
      FROM candidate
      WHERE cand_lname = 'Haven')
   WHERE intrv_num = :intrv_num;
```

In the preceding example, the **LOCOPY** function returns a handle value for the copy of the CLOB **resume** column in the **candidate** table. Because the **LOCOPY** function specifies a table and column name, this new CLOB value has the storage characteristics of this **resume** column. If you omit the table (**candidate**) and column (**resume**) names, the **LOCOPY** function uses the system-defined storage defaults for the new CLOB value. The UPDATE statement then assigns this new CLOB value to the **resume** column in the **interviews** table.

In the following example, the **LOCOPY** function executes on the local database and returns a handle value on the local server for the copy of the BLOB **cand_pic** column in the **election2008** table in **rdb**, which is another database of the local database server. The INSERT statement then assigns this new BLOB value to the **cand_photo** column in the local **candidate** table.

```
INSERT INTO candidate (cand_photo)
   SELECT LOCOPY(cand_pic) FROM rdb:election2008;
```

When the **LOCOPY** function executes on the same database server as the original BLOB or CLOB column in a distributed query, it produces two copies of the BLOB or CLOB value, one in the remote database and the other in the local database, as the following two examples show.

In the first example, the **LOCOPY** function executes on the remote **rdb** database and returns a handle value in the remote database for the copy of the BLOB **cand_pic** column in the remote **election2008** table. The INSERT statement then assigns this new BLOB value to the **cand_photo** column in the local **candidate** table:

```
INSERT INTO candidate (cand_photo)
   SELECT rdb:LOCOPY(cand_pic)
      FROM rdb:election2008;
```

In the second example, the **LOCOPY** function executes on the local database and returns a handle value on the local database for the copy of the BLOB **cand_photo** column in the local **candidate** table. The INSERT statement then assigns this new BLOB value to the **cand_pic** column in the **election2008** table in the remote **rdb** database:

```
INSERT INTO rdb:election2008 (cand_pic)
   SELECT LOCOPY(cand_photo) FROM candidate;
```

The BLOB and CLOB arguments of the built-in **LOCOPY** function are built-in opaque data types. These can be values returned by cross-database DML operations or by cross-database function calls, but built-in opaque types do not support distributed operations across database server instances. If the local database and the **rdb** database are databases of different Informix® instances, the INSERT statements in the previous two examples fail with error -999.

Time Functions

The time functions of Informix® accept DATE or DATETIME arguments, or character representation of a DATE or DATETIME value. They typically return DATE or DATETIME values, or convert information that they extract from DATE or DATETIME values into character strings or integers.

See also the descriptions of the **ROUND** and **TRUNC** functions, which can change the precision of DATE or DATETIME values, in the section Algebraic Functions on page 1057.

Time Functions

" { **DATE**(*non_date_expr*) | { **DAY** | **LAST_DAY** | **MONTH** | **QUARTER** | **WEEKDAY** | **YEAR** } (*date/dtime_expr*) | **EXTEND**(*date/dtime_expr* [, *first***TO** *last*]) | **MDY**(*month* , *day*, *year*) | { **ADD_MONTHS**(*date/dtime_expr* , *integer*) | **MONTHS_BETWEEN**(*date/dtime_expr* , *date/dtime_expr*) | **NEXT_DAY**(*date/dtime_expr* , *weekday*) | { **TO_CHAR**({ *date/dtime_expr* | *num_expr* } | **TO_DATE**(*char_expression* } [, *format_string*]) } } "

Element	Description	Restrictions	Syntax
char_expression	Expression to be converted to a DATE or DATETIME value	Must be a literal, host variable, expression, or column of a character data type	Expression on page 1003

Element	Description	Restrictions	Syntax
date/dtime _expr	Expression that returns a DATE or DATETIME value	Can be host variable, expression, column, or constant.	Expression on page 1003
day	Expression that returns the number of a day of the month	Must return integer > 0 but no greater than the number of days in the specified month	Expression on page 1003
first	Largest time unit in the result. If you omit *first* and *last*, the default *first* is YEAR.	Must be a DATETIME qualifier keyword that specifies a time unit no smaller than *last*	#unique_162
format_string	String that contains a format mask for the first argument	Must be a character data type that specifies a valid format. Can be a column, host variable, expression, or constant	Quoted String on page 1202
integer	Expression that specifies a whole number of months	Must evaluate to positive or negative integer	Expression on page 1003
last	Smallest time unit in the result	Must be a DATETIME qualifier keyword that specifies a time unit no smaller than *first*	#unique_162
month	Expression that represents the number of the month	Must evaluate to an integer in the range from 1 to 12, inclusive	Expression on page 1003
non _date_expr	Expression that represents a value to be converted to a DATE data type	Typically an expression that returns a CHAR, DATETIME, or INTEGER value that can be converted to a DATE data type	Expression on page 1003
num_expr	Expression that evaluates to a real number	Must return a numeric data type	Expression on page 1003
weekday	Abbreviated name of a day of the week	A character data type containing a valid abbreviation for a day of the week	Quoted String on page 1202
year	Number expression that represents a year	Must evaluate to a 4-digit integer. You cannot use a 2-digit abbreviation.	Expression on page 1003

DATE Function

The **DATE** function converts its argument to a DATE value.

Its non-DATE argument can be any expression that can be converted to a DATE value, usually a CHAR, DATETIME, or INTEGER value. The following WHERE clause specifies a quoted string as its CHAR argument:

```
WHERE order_date < DATE('12/31/07')
```

When the **DATE** function interprets a CHAR non-DATE expression, it expects this expression to conform to any DATE format that the **DBDATE** environment variable specifies. For example, suppose **DBDATE** is set to `Y2MD/` when you execute the following query:

```
SELECT DISTINCT DATE('02/01/2008') FROM ship_info;
```

This SELECT statement generates an error, because the **DATE** function cannot convert this string expression. The **DATE** function interprets the first part of the date string (`02`) as the year and the second part (`01`) as the month.

For the third part (`2008`), the **DATE** function encounters four digits when it expects a two-digit day (valid day values must be between `01` and `31`). It therefore cannot convert the value. For the SELECT statement to execute successfully with the `Y2MD/` value for **DBDATE**, the argument would need to be '08/02/01'. For information on the format of **DBDATE**, see the *HCL® Informix® Guide to SQL: Reference*.

For information on the order of precedence among Informix® environment variables that can specify the display and data entry format of DATE values, see the topic Precedence of DATE and DATETIME format specifications.

When you specify a positive INTEGER value for the non-DATE expression, the **DATE** function interprets this as the number of days after December 31, 1899.

If the integer value is negative, the **DATE** function interprets the value as the number of days before December 31, 1899. The following WHERE clause specifies an INTEGER value for the non-DATE expression:

```
WHERE order_date  <  DATE(365)
```

The database server searches for rows with an **order_date** value less than December 31, 1900 (which is 12/31/1899 plus 365 days).

DAY Function

The **DAY** function takes a DATE or DATETIME argument and returns the day of the month as an integer in the range from 1 to the number of days in the current month.

The following statement fragment calls the **DAY** function with the **CURRENT** function as its argument to compare **order_date** column values to the current day of the month:

```
WHERE DAY(order_date) > DAY(CURRENT)
```

MONTH Function

The **MONTH** function returns an integer corresponding to the *month* portion of its DATE or DATETIME argument.

Like the **DAY**, **YEAR**, **WEEKDAY**, and **QUARTER** built-in time functions, the **MONTH** function extracts information from a single DATE or DATETIME expression that is its argument. The return value is the ordinal position of that *month* within the sequence of months in a calendar year. On September 23, for example, the function expression `MONTH(TODAY)` returns `9`,

The following example returns a number that can range from `1` through `12` to indicate the month when the order was placed:

```
SELECT order_num, MONTH(order_date) FROM orders;
```

QUARTER Function

The **QUARTER** function returns an integer in the range 1 through 4 that corresponds to the *quarter* of the calendar year that includes its DATE or DATETIME argument.

For example, any date in January, February, or March returns the integer `1`.

The argument must be an expression that evaluates to a DATE or DATETIME data type.

Examples of QUARTER function expressions

The following function expression returns `3`, because August is in the third quarter of the year.

```
QUARTER('2014-08-25')
```

The following example returns a number that can range from `1` through `4` to indicate the quarter when the order was placed:

```
SELECT order_num, QUARTER(order_date) FROM orders;
```

The following query includes QUARTER function expressions whose arguments are the **order_date** column and the CURRENT operator. The WHERE clause restricts the result set to qualifying rows with **order_date** values in quarters earlier in the current year than the current quarter:

```
SELECT * FROM orders
   WHERE (QUARTER(order_date) < QUARTER(CURRENT))
          AND YEAR(order_date) = YEAR(CURRENT);
```

During the first quarter, however, this query returns no rows, because there can be no data from a quarter with a value less than the current quarter. That is, there is no quarter of value zero.

WEEKDAY Function

The **WEEKDAY** function accepts a DATE or DATETIME argument, and returns an integer in the range from 0 to 6 that represents the day of the week.

As the return value, zero (0) represents Sunday, one (1) represents Monday, and so on.

The following query returns all the orders that were paid on the same day of the week as the current date:

```
SELECT * FROM orders
   WHERE WEEKDAY(paid_date) = WEEKDAY(CURRENT);
```

YEAR Function

The **YEAR** function takes a DATE or DATETIME argument and returns a four-digit integer that represents the year.

The following example lists orders in which the **ship_date** is earlier than the beginning of the current year:

```
SELECT order_num, customer_num FROM orders
   WHERE year(ship_date) < YEAR(TODAY);
```

Similarly, because a DATE value is a simple calendar date, you cannot add or subtract a DATE value with an INTERVAL value whose *last* qualifier is smaller than DAY. In this case, convert the DATE value to a DATETIME value.

MONTHS_BETWEEN Function

The **MONTHS_BETWEEN** function accepts two DATE or DATETIME expressions as arguments, and returns a signed DECIMAL value that quantifies the interval between those arguments in months, as if *month* were a unit of time.

This function requires two arguments, each of which can be a DATE expression or a DATETIME expression.

The value returned is a DECIMAL data type, representing the difference between the two arguments, expressed as a DECIMAL value based on 31-day units. If the first argument is a point in time later than the second argument, the sign of the returned value is positive. If the first argument is earlier than the second argument, the sign of the returned value is negative. If both arguments are equal, the return value is zero.

If the dates of the arguments are both the same days of a month or are both the last days of a months, the result is a whole number. Otherwise, the fractional portion of the result is calculated, based on a month of 31 days . This fractional part can also include the difference in *hour*, *minute*, and *second* time units, unless both arguments are DATE expressions.

The following query calls the **MONTHS_BETWEEN** function in the Projection clause, using two DATE values returned by **TO_DATE** expressions as arguments.

```
SELECT MONTHS_BETWEEN(TO_DATE('2-2-2005', '%m-%d-%Y'),
                      TO_DATE('1-1-2005',  '%m-%d-%Y'))
AS lunations FROM systables WHERE tabid = 1;
```

The value returned by the query expresses the 32-day difference between the two DATE arguments as a positive number of 31-day months:

```
          months
1.03225806451613
```

The next example returns the DATETIME column expression arguments to **MONTHS_BETWEEN** expressions, and their differences in months for two rows of a table:

```
SELECT d_datetime, e_datetime,
   MONTHS_BETWEEN(d_datetime, e_datetime) AS months_between
   FROM mytab1;


d_datetime      2007-11-01 09:00:00.00000
e_datetime      2007-12-07 14:30:12.12345
months_between  -1.2009453405018


d_datetime      2007-12-13 09:40:30.00000
e_datetime      2007-11-13 08:40:30.00000
months_between  1.00000000000000
```

Here the first **MONTHS_BETWEEN** result includes differences in time units smaller than days. The second result has no fractional part, because the *day* time units of the arguments had the same value.

The **MONTHS_BETWEEN** expressions in the next example compares DATE and DATETIME values:

```
SELECT col_datetime, col_date,
   MONTHS_BETWEEN(col_datetime, col_date) AS months_between
   FROM mytab2;


col_datetime    2008-12-13 08:40:30.00000
col_date        11/13/2007
months_between  13.0000000000000
```

Because both arguments specify the same day of the month, the result has no fractional part.

EXTEND Function

The **EXTEND** function adjusts the precision of a DATETIME or DATE value.

The DATETIME or DATE expression that is its first argument cannot be a quoted string representation of a DATE value.

If you do not specify *first* and *last* qualifiers, the default qualifiers are YEAR TO FRACTION(3).

If the expression contains fields that are not specified by the time-unit qualifiers, those fields are discarded.

If the *first* qualifier specifies a larger (that is, more significant) time unit than what exists in the expression, the new fields are filled in with values returned by the **CURRENT** function. If the *last* qualifier specifies a smaller time unit (that is, less significant) than what exists in the expression, the new fields are filled in with constant values. A missing MONTH or DAY field is filled in with `1`, and the missing HOUR to FRACTION fields are filled in with `0`.

In the following expression, the **EXTEND** call returns the **call_dtime** column value with YEAR TO SECOND precision:

```
EXTEND (call_dtime, YEAR TO SECOND)
```

You can use the **EXTEND** function to perform addition or subtraction with a DATETIME value and an INTERVAL value that do not have the same time unit qualifiers. The next expression expands a literal DATETIME YEAR TO DAY value to a precision of YEAR TO MINUTE so that an interval YEAR TO MINUTE value can be subtracted from it:

```
EXTEND (DATETIME (2009-8-1) YEAR TO DAY, YEAR TO MINUTE)
   - INTERVAL (720) MINUTE (3) TO MINUTE
```

You can use the **EXTEND** function to selectively update a subset of the time units in a DATETIME value. The UPDATE statement in the next example updates only the *hour* and *minute* time unit values in a DATETIME YEAR TO MINUTE column.

```
UPDATE cust_calls SET call_dtime = call_dtime -
   (EXTEND(call_dtime, HOUR TO MINUTE) - DATETIME (11:00)
   HOUR TO MINUTE) WHERE customer_num = 106;
```

Subtracting `11:00` from the DATETIME HOUR TO MINUTE value returned by **EXTEND** yields a positive or negative INTERVAL HOUR TO MINUTE value. Subtracting this difference from the original value in the **call_dtime** column forces the updated *hour* and *minute* time unit values to `11:00` in the **cust_calls.call_dtime** column.

MDY Function

The **MDY** function takes as its arguments three integer expressions that represent the *month*, *day*, and *year*, and returns a type DATE value.

- The first argument represents the number of the month (1 to 12).
- The second argument represents the number of the day of the month (1 to 28, 29, 30, or 31, as appropriate for the month).
- The third argument represents the 4-digit year. You cannot use a 2-digit abbreviation.

Example of the MDY function in an UPDATE statement

The following example updates a row in the **orders** table whose purchase-order number is `8052` by changing its **paid_date** column value to the first day of the current month:

```
UPDATE orders SET paid_date = MDY(MONTH(TODAY), 1, YEAR(TODAY))
   WHERE po_num = '8052';
```

Here the first and last arguments to the **MDY** function are time expressions that return integers corresponding to the current *month* and *year*. The second argument specifies the *day of the month* as a literal integer. Reversing the order of the first two arguments in this example would change the **paid_date** value to some day before January 13 of the current year, unless the same application also included error-checking code that identified that date as too early to be valid.

TO_DATE Function

The **TO_DATE** function converts a character string to a DATETIME value. The function evaluates the *char_expression* first argument as a date, according to the date format that the *format_string* second argument specifies, and returns the equivalent date.

If *char_expression* is NULL, then a NULL value is returned.

Any argument to the **TO_DATE** function must be of a built-in data type.

If you omit the *format_string* parameter, the **TO_DATE** function applies the default DATETIME format to the DATETIME value. The default DATETIME format is specified by the **GL_DATETIME** environment variable.

In the following example, the user wants to convert a character string to a DATETIME value in order to update the **begin_date** column of the **tab1** table with the converted value. The **begin_date** column is defined as a DATETIME YEAR TO SECOND data type. The user uses an UPDATE statement that contains a **TO_DATE** function to accomplish this result:

```
UPDATE tab1
  SET begin_date = TO_DATE('Wednesday July 25, 2007 18:45',
  '%A %B %d, %Y %R');
```

The *format_string* parameter in this example tells the **TO_DATE** function how to format the converted character string in the **begin_date** column. For a table that shows the effect of each format symbol in this format string, see #unique_1590.

TO_NUMBER Function

The **TO_NUMBER** function can convert a number or a character expression representing a number value to a DECIMAL data type.

The **TO_NUMBER** function has this syntax:

TO_NUMBER Function

" **TO_NUMBER**({ *char_expr* | *num_expr* }) "

Element	Description	Restrictions	Syntax
char _expression	Expression to be converted to a DECIMAL value	Must be a literal, host variable, expression, or column of a character data type	Expression on page 1003
num_expression	Expression that evaluates to a real number	Must return a numeric data type	Expression on page 1003

The **TO_NUMBER** function converts its argument to a DECIMAL data type. The argument can be the character string representation of a number or a numeric expression.

The following example retrieves a DECIMAL value that the **TO_NUMBER** function returns from the literal representation of a MONEY value:

```
SELECT TO_NUMBER('$100.00') from mytab;
```

The following table shows the output of this SELECT statement.

(expression)
`100.000000000000`

In this example, the currency symbol is discarded from the `'$100.00'` string.

The **TO_NUMBER** function is not required in most contexts, because by default, Informix® converts numbers that include a decimal point (and quoted strings in the format of a literal number that has a decimal point) to a DECIMAL data type. This function can be useful, however, when you are migrating SQL applications that were originally written for other database servers, if the application makes calls to a function of this name that returns a DECIMAL value.

Trigonometric Functions

The built-in trigonometric functions calculate ratios of the lengths of the sides of right triangles. Two supporting functions, **DEGREES** and **RADIANS**, can respectively convert the units of angular values from radians to degrees, and from degrees to radians.

The built-in trigonometric functions have the following syntax.

Trigonometric Functions

" { { **COS** | **COSH** | **SIN** | **SINH** | **TAN** | **TANH** | **DEGREES** } (*radian_expr*) | { { **ASIN** | **ASINH** | **ACOS** | **ACOSH** | **ATAN** | **ATANH** } (*numeric_expr*) | **ATAN2** (*y*, *x*) | **RADIANS** (*degree_expr*) } } "

Element	Description	Restrictions	Syntax
degree_expr	Expression that represents the number of degrees	Must return a value that can be converted to a DECIMAL type	Expression on page 1003

Element	Description	Restrictions	Syntax
numeric_expr	Expression that serves as the argument to the **ASIN**, **ACOS**, **ATAN**, **ASINH**, **ACOSH** or **ATANH** functions	Must return a value between -1 and 1, inclusive	Expression on page 1003
radian_expr	Expression that represents the number of radians	Must return a numeric value	Expression on page 1003
x	Expression that represents the x coordinate in the rectangular coordinate pair (*x*, *y*)	Must return a numeric value	Expression on page 1003
y	Expression that represents the y coordinate in the rectangular coordinate pair (*x*, *y*)	Must return a numeric value	Expression on page 1003

Sections that follow describe each of these built-in trigonometric functions.

COS Function

The **COS** function returns the cosine of a radian expression.

The following example returns the cosine of the values of the degrees column in the **anglestbl** table. The expression passed to the **COS** function in this example converts degrees to radians.

```
SELECT COS(degrees*180/3.1416) FROM anglestbl;
```

COSH function

The **COSH** function returns the hyperbolic cosine of the required argument, where the argument is an angle expressed in radians.

COSH function

" **COSH**(*radian_expr*) "

Element	Description	Restrictions	Syntax
radian_expr	Expression that evaluates to an angular value in units of radians	Must be of a numeric data type	Expression on page 1003

The following example returns the hyperbolic cosine of the values in the degrees column of the **anglestbl** table. The expression passed to the **COSH** function converts the degrees to radians.

```
SELECT COSH(degrees*180/3.1416) FROM anglestbl;
```

SIN Function

The **SIN** function returns the sine of an angle that you specify as its radian expression argument.

The following query returns the sines of the values in each row of the **radians** column of the **anglestbl** table:

```
SELECT SIN(radians) FROM anglestbl;
```

SINH function

The SINH function returns the hyperbolic sine of the argument, where the argument is an angle expressed in radians.

SINH Function

" `SINH`(*radian_expr*) "

The following example returns the hyperbolic sine of the values in the degrees column of the **anglestbl** table. The expression passed to the SINH function converts the degrees to radians.

```
SELECT SINH(degrees*180/3.1416) FROM anglestbl;
```

TAN Function

The **TAN** function returns the value of the tangent of its radian expression argument.

This example returns the tangent of the values in the **radians** column of the **anglestbl** table:

```
SELECT TAN(radians) FROM anglestbl;
```

TANH Function

The TANH function returns the hyperbolic tangent of the argument, where the argument is an angle expressed in radians.

TANH Function

" `TANH`(*radian_expr*) "

" "

The following example returns the hyperbolic tangent of the values in the degrees column of the **anglestbl** table. The expression passed to the TANH function converts the degrees to radians.

```
SELECT TANH(degrees*180/3.1416) FROM anglestbl;
```

ACOS Function

The **ACOS** function returns the arc cosine of a numeric expression.

The following example returns the arc cosine of the value (-0.73) in radians:

```
SELECT ACOS(-0.73) FROM anglestbl;
```

ACOSH Function

The ACOSH function returns the hyperbolic tangent of the specified numeric input.

ACOSH Function

" `ACOSH(`*numeric_expr*`)` "

" "

ASIN Function

The **ASIN** function returns the arc sine of a numeric expression argument.

The following example returns the arc sine of the value (-0.73) in radians:

```
SELECT ASIN(-0.73) FROM anglestbl;
```

ASINH Function

The ASINH function returns the arc hyperbolic sine of the specified numeric input.

ASINH Function

" `ASINH(`*numeric_expr*`)` "

" "

ATAN Function

The **ATAN** function returns the arc tangent of a numeric expression.

The following example returns the arc tangent of the value (-0.73) in radians:

```
SELECT ATAN(-0.73) FROM anglestbl;
```

ATANH Function

The ATANH function returns the hyperbolic tangent of the specified numeric input.

ATANH Function

" ATANH(*numeric_expr*) "

" "

ATAN2 Function

The **ATAN2** function computes the angular component of the polar coordinates (*r*, q) associated with (*x, y*).

The following example compares *angles* to q for the rectangular coordinates (4, 5):

```
WHERE angles > ATAN2(4,5)     --determines q for (4,5) and
                                --compares to angles
```

You can determine the length of the radial coordinate *r* using the expression that the following example shows:

```
SQRT(POW(x,2) + POW(y,2))     --determines r for (x,y)
```

You can determine the length of the radial coordinate *r* for the rectangular coordinates (4,5) using the expression that the following example shows:

```
SQRT(POW(4,2) + POW(5,2))     --determines r for (4,5)
```

DEGREES function

Use the **DEGREES** function to convert the value of an expression or host variable representing a number of radians to the equivalent number of degrees.

The *radian_expression* or host variable that is the only argument to this function must be a numeric data type (or a non-numeric data type that can be converted to a number) that the database server evaluates in units of radians, and converts to units of degrees.

The return value is a number of type DECIMAL (32, 255).

In both of the examples that follow, the argument to **DEGREES** evaluates to 6 radians, and the return value is 343.774677078494 degrees:

```
EXECUTE FUNCTION DEGREES (6);
EXECUTE FUNCTION DEGREES ("6");
```

The **DEGREES** function converts radians to degrees according to the following formula:

```
(number of radians) * (180/pi) = (number of degrees)
```

Here **pi** represents the ratio of the circumference of a circle to its diameter. Results of arithmetic calculations that use the transcendental number **pi** as the divisor of a rational number always include rounding error.

RADIANS function

Use the **RADIANS** function to convert an expression or a host variable representing a number of degrees to the equivalent number of radians.

The return value is a number of type DECIMAL (32, 255).

The *degree_expression* that is the only argument to this function must have a numeric data type (or a non-numeric data type that can be converted to a number) that the database server evaluates in units of degrees, and converts to units of radians:

```
EXECUTE FUNCTION RADIANS (100);
EXECUTE FUNCTION RADIANS ("100");
```

In both of the examples above, the **RADIANS** argument evaluates to 100 degrees, and the return value is 1.745328251994 radians. You can use a **RADIANS** function expression as the argument to the **COS**, **SIN**, or **TAN** function to return the respective trigonometric values for that angle:

```
COS(RADIANS (100))
SIN(RADIANS ("100"))
TAN(RADIANS (100))
```

The **RADIANS** function converts degrees to radians according to the following formula:

```
(number of degrees) * (pi/180) = (number of radians)
```

Here **pi** represents the ratio of the circumference of a circle to its diameter. Results of arithmetic calculations that use the transcendental number **pi** as the dividend of a rational number always include rounding error.

String-Manipulation Functions

String-manipulation functions perform various operations on strings of characters.

The string-manipulation functions are identified in the following diagram:

(explicit id unique_1445_Connect_42_smfu001) unique_1445_Connect_42_smfu001(explicit id unique_1445_Connect_42_smfu002)
unique_1445_Connect_42_smfu002(explicit id unique_1445_Connect_42_smfu003) unique_1445_Connect_42_smfu003(explicit
id unique_1445_Connect_42_smfu004) unique_1445_Connect_42_smfu004(explicit id unique_1445_Connect_42_smfu005)
unique_1445_Connect_42_smfu005(explicit id unique_1445_Connect_42_smfu006) unique_1445_Connect_42_smfu006(explicit
id unique_1445_Connect_42_smfu007) unique_1445_Connect_42_smfu007(explicit id unique_1445_Connect_42_smfu008)
unique_1445_Connect_42_smfu008(explicit id unique_1445_Connect_42_smfu009) unique_1445_Connect_42_smfu009(explicit
id unique_1445_Connect_42_smfu010) unique_1445_Connect_42_smfu010(explicit id unique_1445_Connect_42_smfu011)
unique_1445_Connect_42_smfu011(explicit id unique_1445_Connect_42_smfu015) unique_1445_Connect_42_smfu015(explicit id
unique_1445_Connect_42_smfu012) unique_1445_Connect_42_smfu012(explicit id unique_1445_Connect_42_smfu013) unique_1445_Connect_42_smfu013

String-Manipulation Functions

"{ <CONCAT Function> | <ASCII Function> | <TRIM Function> | <LTRIM Function> | <RTRIM Function> | <SPACE Function> | <REVERSE Function> | <REPLACE Function> | <LPAD Function> | <RPAD Function> | <CHR Function> | <Case-Conversion Functions> | <Substring Functions> }"

Sections that follow describe each of the built-in string manipulation functions.

CONCAT Function

The **CONCAT** function accepts two expressions as arguments, and returns a single character string that appends the string representation of the value returned by its second argument to the string representation of the value returned by its first argument.

CONCAT Function

"**CONCAT**(*expr_1*, *expr_2*)"

Element	Description	Restrictions	Syntax
expr_1, *expr_2*	Expressions whose string representations of their values are to be concatenated	Cannot return a complex, user-defined, or large object type. If a host variable, it must be long enough to store the resulting combined strings.	Expression on page 1003

Each arguments to the **CONCAT** function can evaluate to a character, number, or time data type. If either or both of the concatenated arguments is null, the function returns a NULL value.

Unlike other built-in string manipulation functions of Informix®, the **CONCAT** function cannot be overloaded.

CONCAT is the operator function of the concatenation (||) operator, For a given pair of expression arguments, **CONCAT** returns the same string as that operator returns from the same expressions as operands. See Concatenation Operator on page 1021 for additional information about concatenation operations, and for restrictions on the SQL and Dynamic SQL statements in which you can invoke the **CONCAT** function.

Return Types from CONCAT and String Functions

The data type of the return value from a successful call to the **CONCAT** function (or from the concatenation (||) operator, or from a call to other built-in string-manipulation functions that follow the same rules as **CONCAT** for determining their return type) depends on the data types of the arguments and on the length of the resulting string. The order of the two arguments is not significant in determining the return type.

Informix® applies the following rules for the return type from operations that concatenate values that arguments of more than one data type specify:

- If one of the types is National Language Support (namely NCHAR and NVARCHAR):
  - the return type is NVARCHAR if the resulting length is less than 255 bytes
  - the return type is NCHAR otherwise.
- If one of the arguments is VARCHAR or a number type,
  - the return type is VARCHAR if the resulting length is less than 255 bytes
  - the return type is LVARCHAR otherwise.
- An exception to these rules, however, can occur in certain cross-server operations in which a remote routine is executed locally, and a concatenation expression is evaluated locally before its return value is sent to a remote database server. For remote servers that do not support the LVARCHAR data type in distributed transactions, the concatenated result is sent as a CHAR data type if sending the LVARCHAR type returns an error. Informix® database server instances earlier than Version 11.10 require a CHAR return value in this scenario.

In the following table, the rows list the valid data types of the first argument to the **CONCAT** function, and the columns list the type of the second argument. The cell at the intersection of each row and column shows the possible returned type or types. The row and the column labelled as **Other** represent arguments that evaluate to non-character types, such as number or time data types like DECIMAL or DATE.

Table 26. Return Types from Operations on Two Arguments (in Version 11.50.xC2 or Later)

	NCHAR	NVARCHAR	CHAR	VARCHAR	LVARCHAR	Other
NCHAR	nchar	nvarchar *or* nchar	nchar	nvarchar *or* nchar	nvarchar *or* nchar	nvarchar *or* nchar
NVARCHAR	nvarchar *or* nchar	nvarchar *or* nchar	nvarchar *or* nchar	nvarchar *or* nchar	nvarchar *or* nchar	nvarchar *or* nchar
CHAR	nchar	nvarchar *or* nchar	char	varchar *or* lvarchar	lvarchar	varchar *or* lvarchar

Table 26. Return Types from Operations on Two Arguments (in Version 11.50.xC2 or Later) (continued)

	NCHAR	NVARCHAR	CHAR	VARCHAR	LVARCHAR	Other
VARCHAR	nvarchar *or* nchar	nvarchar *or* nchar	varchar *or* lvarchar	varchar *or* lvarchar	lvarchar	varchar *or* lvarchar
LVARCHAR	nvarchar *or* nchar	nvarchar *or* nchar	lvarchar	lvarchar	lvarchar	lvarchar
Other	nvarchar *or* nchar	nvarchar *or* nchar	varchar *or* lvarchar	varchar *or* lvarchar	lvarchar	varchar *or* lvarchar

For string manipulation functions other than **CONCAT**, arguments of DATE, DATETIME, or MONEY data types always return an NVARCHAR or NCHAR value, depending on the length of the resulting string.

This table is symmetrical, because the order of arguments has no affect on the return data type. User-defined data types, large-object types, complex types, and other extended data types are not valid as arguments to the built-in string-manipulation functions or operators.

This table also describes the return data types of expressions that use the concatenation (||) operator.

Not shown here is the result of concatenation operations in which the sum of the argument lengths exceeds the approximately 32Kb limit for CHAR, NCHAR, and LVARCHAR data types. This returns error -881, rather than a concatenated data value. Because the maximum LVARCHAR size is 32.739 bytes, and the CHAR and NCHAR limits are both 32,767 bytes, error -881 is usually associated with VARCHAR and NVARCHAR objects, whose limit is 255 bytes, but automatic return type promotion can reduce the incidence of this error.

The following string-manipulation functions support the same rules as **CONCAT** for return type promotion:

- **LPAD**
- **RPAD**
- **REPLACE**
- **SUBSTR**
- **SUBSTRING**
- **TRIM**
- **LTRIM**
- **RTRIM**

The following table summarizes how Informix® determines the return type from these string manipulating functions, based on the argument types:

Table 27. String Manipulation Functions that Support Return Type Promotion

Function	How the Return Type of the Function is Determined
CONCAT, ||	Return type is based on both arguments. Refer to Table 26: Return Types from Operations on Two Arguments (in Version 11.50.xC2 or Later) on page 1115..

Table 27. String Manipulation Functions that Support Return Type Promotion (continued)

Function	How the Return Type of the Function is Determined
SUBSTR, **SUBSTRING**	Return type is the same as the *source string* type. If *source string* is a host variable, the return type is NVARCHAR or NCHAR, depending on the length of the result.
TRIM, **LTRIM**, **RTRIM**	Return type depends on the source type and the returned length: • NVARCHAR *returns* NVARCHAR • VARCHAR *returns* VARCHAR • CHAR *returns* VARCHAR (if length <= 255 bytes) • CHAR *returns* LVARCHAR (if length > 255 bytes) • NCHAR *returns* NVARCHAR (if length <= 255 bytes) • NCHAR *returns* LVARCHAR (if length > 255 bytes) • LVARCHAR *returns* LVARCHAR
LPAD, **RPAD**	Return type is based on the *source_string* and *pad_string* arguments. If *pad_string* is not specified, the return type is based on the data type of *source_string*.
REPLACE	Return type is based on the *source_string* and *old_string* arguments (and on the *new_string* argument, if that is specified). If any argument is a host variable, the return type is NCHAR.
ENCRYPT_AES, **ENCRYPT_TDES**, **DECRYPT_BINARY**, **DECRYPT_CHAR**,	For arguments that are not BLOB or CLOB variables, the return type is based on the data types of the *data* and *encrypted_data* arguments. Refer to Table 26: Return Types from Operations on Two Arguments (in Version 11.50.xC2 or Later) on page 1115.

Data-type promotion in NLSCASE INSENSITIVE databases

In databases that have the NLSCASE INSENSITIVE property, the database server disregards the lettercase of NCHAR and NVARCHAR values. Expressions in which functions or operators avoid overflow errors by performing an implicit cast can produce different results from what a case-sensitive database would return, if the expression evaluates to an NCHAR or NVARCHAR data type.

When a string function or a string operator on which the database server supports data-type promotion returns a value that would produce an overflow error for the default VARCHAR or NVARCHAR data type of the expression, the database server performs an implicit cast on the return value, as indicated in the first table of the topic Return Types from CONCAT and String Functions on page 1115:

- If the none of the arguments or operands are NCHAR or NVARCHAR data types, the expression evaluates to a CHAR, LVARCHAR, or VARCHAR data type.
- If any argument or operand is an NCHAR or NVARCHAR data type, the expression evaluates to an NCHAR or NVARCHAR data type.

In databases that have the NLSCASE INSENSITIVE property, operations on CHAR, LVARCHAR, or VARCHAR data types are case-sensitive, but operations on NCHAR or NVARCHAR data types are case-insensitive. Data-type promotion also produces case-insensitive results (rather than case-sensitive) from evaluating an expression that includes CHAR, LVARCHAR, or VARCHAR components, if the same expression also includes NCHAR or NVARCHAR character strings.

The following example illustrates this behavior in a NLSCASE INSENSITIVE database, in which table **t1** has a character column of each of the five built-in character data types. The table stores three rows, in which each column stores the same lettercase variants of a 3-letter character string:

```
CREATE DATABASE db NLSCASE INSENSITIVE;
CREATE TABLE t1 (
   c1 NCHAR(20),
   c2 NVARCHAR(20),
   c3 CHAR((20),
   c4 VARCHAR(20),
   c5 LVARCHAR(20)) ;
INSERT INTO t1 values ('ibm', 'ibm', 'ibm', 'ibm', 'ibm');
INSERT INTO t1 values ('Ibm', 'Ibm', 'Ibm', 'Ibm', 'Ibm');
INSERT INTO t1 values ('IBM', 'IBM', 'IBM', 'IBM', 'IBM');
```

The following query retrieves the values from an NCHAR column, using an equality predicate for a literal string whose letters are all lowercase:

```
SELECT c1 FROM t1 WHERE c1 = 'ibm';
```

Because NCHAR values are not case sensitive in this database, the query returns the column **c1** value from every row:

```
c1

ibm
Ibm
IBM
```

The following query on the same table returns the same case-insensitive results from CHAR column **c3** that the WHERE clause casts to an NCHAR value:

```
SELECT c1 FROM t1 WHERE c3 = 'ibm'::NCHAR(10);
```

After the cast, the **c3** values become case insensitive, so that every row in **c3** matches the string `'ibm'`, and the WHERE condition is true for every row in **c1** :

```
c1

ibm
Ibm
IBM
```

Because case-insensitive operations disregard differences in letter case among strings where the same letters appear in the same sequence, as in the previous example, care must be taken in databases that have the NLSCASE INSENSITIVE property to avoid contexts where data type-promotion applies case-insensitive rules to operations that you expected to be case sensitive.

See also the section Duplicate rows in NLSCASE INSENSITIVE databases on page 699.

Return String Types in Distributed Transactions

In cross-database distributed queries that access tables in different databases of the same Informix®Informix® instance, the same types are returned by **CONCAT** (and by other built-in string manipulation functions that follow the same rules for return type promotion) that the section Return Types from the CONCAT Function on page 1115 describes.

The same types are also returned in cross-server distributed queries, if every participating Informix® instance that evaluates these string-manipulation function expressions is no earlier than Version 11.50.xC2.

For cross-server distributed operations in which the return value is evaluated on a remote Informix® instance earlier than Version 11.50.xC2, the following table (in the same format as the table for Version 11.50.xC2 and later) lists the possible return data types (or -881 overflow error) for the specified data types of arguments to the string-manipulation function (or for operands of the concatenation (||) operator:

Table 28. Return Types from Distributed Operations (in Version 11.50.xC1 and earlier)

	NCHAR	NVARCHAR	CHAR	VARCHAR	LVARCHAR	Other
NCHAR	nchar	nvarchar *or* EM -881	nchar	nchar	nchar	nchar
NVARCHAR	nchar *or* EM -881	nvarchar *or* EM -881	nvarchar *or* EM -881	nvarchar *or* EM -881	nvarchar *or* EM -881	nvarchar *or* EM -881
CHAR	nchar	nvarchar *or* EM -881	char	varchar *or* EM -881	char	char
VARCHAR	nchar	nvarchar *or* EM -881	varchar *or* EM -881	varchar *or* EM -881	varchar *or* EM -881	varchar *or* EM -881
LVARCHAR	nchar	nvarchar *or* EM -881	char	varchar *or* EM -881	char	char
Other	nchar	nvarchar *or* EM -881	char	varchar *or* EM -881	char	char

The following are among the differences between the return values in this release and what Informix® versions earlier than 11.50.xC2 return:

- Earlier releases accept LVARCHAR arguments, but cannot return LVARCHAR values.
- If the result is longer than the maximum size of the argument of the longest data type, earlier releases do not support data-type promotion, but issue error -881. (This is typically with VARCHAR or NVARCHAR arguments, if the length of the resulting string would be greater than 255 bytes.)

For all versions of Informix®, error -881 is issued if the length of a returned string exceeds 32Kb.

ASCII Function

The **ASCII** function returns the decimal representation of the first character in a character string, based on its codepoint in the ASCII character set.

ASCII Function

" `ASCII(`*char_expr*`)` "

Element	Description	Restrictions	Syntax
char_expr	Expression that evaluates to a character data type	Must be of type CHAR, LVARCHAR, NCHAR, NVARCHAR, or VARCHAR	Identifier on page 1231

The **ASCII** function takes a single argument of any character data type. It returns an integer value, based on the first character of the argument, corresponding to the decimal representation of the codepoint of that character within the ASCII character set.

If the argument is NULL, or if the argument is an empty string, the **ASCII** function returns a NULL value.

The following query returns the ASCII value of uppercase `H`:

```
SELECT ASCII("HELLO") FROM systables WHERE tabid = 1;
```

The following table shows the output of this SELECT statement.

(constant)
72

The following query returns the ASCII value of lowercase `h`:

```
SELECT ASCII("hello") FROM systables WHERE tabid = 1;
```

The following table shows the output of this SELECT statement.

(constant)
104

The following query returns the **ASCII** output from an empty string argument:

```
SELECT ASCII("") FROM systables WHERE tabid = 1;
```

The following table shows the NULL output of this SELECT statement.

(constant)

The following query returns the **ASCII** output from a NULL argument:

```
SELECT ASCII(NULL) FROM systables WHERE tabid = 1;
```

The following table shows the NULL output of this SELECT statement.

(constant)

The **ASCII** function interprets this argument as a NULL expression, rather than as a value that begins with uppercase N.

For a table of the numeric values of the codepoints in the ASCII character set, see #unique_1071.

TRIM Function

The **TRIM** function removes specified leading or trailing pad characters from a string. (See also the descriptions of the **LTRIM** and **RTRIM** functions, which provide similar functionality, but support a different syntax.)

TRIM Function

"**TRIM**"

"([{ **BOTH** | **TRAILING** | **LEADING** } [*pad_char*] **FROM**] *source_expression*)"

Element	Description	Restrictions	Syntax
pad_char	Expression that evaluates to a single character or NULL. The default is a blank space (= ASCII 32).	Must be a character expression	Expression on page 1003
source_expression	Character expression, including a character column name, or a call to another **TRIM** function	Cannot be a DISTINCT data type	Expression on page 1003

The **TRIM** function returns a character string identical to its *source_expression* argument, except that any leading or trailing pad characters, as specified by the LEADING, TRAILING, or BOTH keywords, are deleted. If no trim qualifier (LEADING, TRAILING, or BOTH) is specified, BOTH is the default. If no *pad_char* is specified, a single blank space (the ASCII 32 character) is the default, and leading or trailing blank spaces, as specified by the qualifying keyword, are deleted from the returned value.

If either the *pad_char* or the *source_expression* evaluates to NULL, the result of the **TRIM** function is NULL.

The data type of the returned value depends on the *source_expression* argument:

- If the argument is longer than 255 bytes, the returned value is of type LVARCHAR.
- If the argument has 255 bytes or fewer, the data type of the returned value depends on the data type of the argument:

- If the argument is of type CHAR or VARCHAR, a VARCHAR value is returned.
- If the argument is of type NCHAR or NVARCHAR, an NVARCHAR value is returned.
- If the argument is of type LVARCHAR, an LVARCHAR value is returned.

The length of the returned value is 255 bytes or fewer for VARCHAR or NVARCHAR *source_expression* arguments, and no more than 32,739 bytes for CHAR, NCHAR, or LVARCHAR arguments.

The following example shows some generic uses for the **TRIM** function:

```
SELECT TRIM (c1) FROM tab;
SELECT TRIM (TRAILING '#' FROM c1) FROM tab;
SELECT TRIM (LEADING FROM c1) FROM tab;
UPDATE c1='xyz' FROM tab WHERE LENGTH(TRIM(c1))=5;
SELECT c1, TRIM(LEADING '#' FROM TRIM(TRAILING '%' FROM
   '###abc%%%')) FROM tab;
```

In Dynamic SQL, when you use the DESCRIBE statement with a SELECT statement that calls the **TRIM** function in the Projection list, the data type of the trimmed column that DESCRIBE returns depends on the data type of the *source_expression*, for SQL data type constants defined in the **sqltypes.h** header file of the source file. For further information on the GLS aspects of the **TRIM** function in , see the *HCL® Informix® GLS User's Guide*.

Fixed Character Columns

The **TRIM** function can be specified on fixed-length character columns. If the length of the string is not completely filled, the unused characters are padded with blank space. Figure 14: Column Entry in a Fixed-Length Character Column on page 1122 shows this concept for the column entry '`##A2T##`', where the column is defined as CHAR(10).

Figure 14. Column Entry in a Fixed-Length Character Column

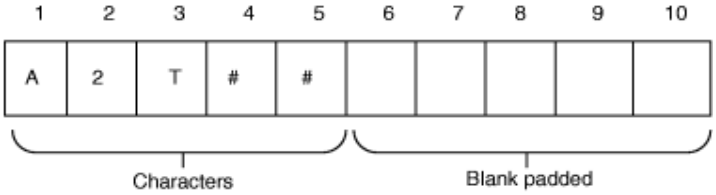


If you want to trim the sharp sign (#) *pad_char* from the column, you need to consider the blank padded spaces as well as the actual characters.

For example, if you specify the keyword BOTH, the result from the trim operation is `A2T##`, because the **TRIM** function does not match the blank padded space that follows the string. In this case, the only sharp signs (#) trimmed are those that precede the other characters. The SELECT statement is shown, followed by Figure 15: Result of TRIM Operation on page 1123, which presents the result.

```
SELECT TRIM(LEADING '#' FROM col1) FROM taba;
```

Figure 15. Result of TRIM Operation



This SELECT statement removes all occurrences of the sharp (#) sign:

```
SELECT TRIM(BOTH '#' FROM TRIM(TRAILING ' ' FROM col1)) FROM taba;
```

LTRIM Function

The **LTRIM** function removes specified leading pad characters from a string.

LTRIM Function

" **LTRIM**(*source_string* [,*pad_string*]) "

Element	Description	Restrictions	Syntax
pad_string	Expression that specifies one or more characters to delete from *source_string*	Must be a character expression	Expression on page 1003
source_string	Expression that specifies a character string from which characters in *pad_string* are deleted	Pad characters to the right of any character not in *pad_string* are not deleted	Expression on page 1003

The first argument to the **LTRIM** function must be a character expression from which to delete leading pad characters. The optional second argument is a character expression that evaluates to a string of pad characters. If no second argument is provided, only blank characters are regarded as pad characters.

The return data type of the **LTRIM** function is based on its source_string argument, using the return type promotion rules that the section Return Types from the CONCAT Function on page 1115 describes.

The value returned contains a substring of *source_string*, but from which any leading pad characters to the left of the first non-pad character have been removed. If a host variable is used, an LVARCHAR data type is returned.

The **LTRIM** function scans a copy of the *source_string* from the left, deleting any leading characters that appear in the *pad_string*. If no *pad_string* argument is specified, only leading blanks are deleted from the returned value. When the first non-pad character is encountered, the function returns its result string and terminates.

In the following example, the *pad_string* is `'Hello'`:

```
SELECT LTRIM('Hellohello world!', 'Hello') FROM mytab;
```

The following table shows the output of this SELECT statement.

(constant)
hello world!

Here the first five characters of the *source_string* were dropped because they matched characters in the *pad_string* , but the function terminated after it encountered the lowercase `h` character, which preserved the trailing `'ello'` pad characters to its right.

RTRIM Function

The **RTRIM** function removes specified trailing pad characters from a string.

RTRIM Function

" **RTRIM**(*source_string* [, *pad_string*]) "

Element	Description	Restrictions	Syntax
pad_string	Expression that specifies one or more characters to delete from *source_string*	Must be a character expression	Expression on page 1003
source_string	Expression that specifies a character string from which characters in *pad_string* are deleted	Pad characters to the left of any character not in *pad_string* are not deleted	Expression on page 1003

The first argument to the **RTRIM** function must be a character expression from which to delete trailing pad characters. The optional second argument is a character expression that evaluates to a string of pad characters. If no second argument is provided, only blank characters are regarded as pad characters.

The return data type of the **LTRIM** function is based on its *source_string* argument, using the return type promotion rules that the section Return Types from the CONCAT Function on page 1115 describes.

The value returned contains a substring of *source_string*, but from which any trailing pad characters to the right of the first non-pad character have been removed. If a host variable is used, an LVARCHAR data type is returned.

The **RTRIM** function scans a copy of the *source_string* from the right, deleting any trailing characters that appear in the *pad_string*. If no *pad_string* argument is specified, only trailing blanks are deleted from the returned value. When the first non-pad character is encountered, the function returns its result string and terminates.

In the following example, the *pad_string* is `' theend!*#?'`:

```
SELECT RTRIM('good night...   *!#?theend ', ' theend!*#?') AS closing FROM mytab;
```

The following table shows the output of this SELECT statement.

(constant)
`good night...`

Here the last fifteen characters of the *source_string* were dropped because they matched characters in the *pad_string*, but the function terminated after it encountered the period (`.`) characters, which preserved the leading `'thn'` pad characters to the left.

SPACE function

The SPACE function creates a character string of a specified number of blank spaces. The maximum length of the returned string value can be 32,739 blank characters.

The function has this syntax:

SPACE Function

" **SPACE**(*expression*) "

Element	Description	Restrictions	Syntax
expression	Expression that evaluates to a non-negative whole number < 256	Must be an expression, constant, column, or host variable of a built-in integer type, or one that can be converted to an integer	Expression on page 1003

The argument to the SPACE function must be of a built-in data type.

The SPACE function returns an LVARCHAR string of the specified number of blank (ASCII 32) characters.

If the argument evaluates to a NULL value, or to a number less than 1, this function returns a NULL value, rather than an empty string.

In the following example, the SPACE function returns a single-character blank string:

```
SELECT SPACE(1) FROM tabula_rasa;
```

The following table shows the output from this SELECT statement, which is a single blank character:

(constant)

REVERSE function

The **REVERSE** function accepts a character expression as its argument, and returns a string of the same length, but with the ordinal positions of every logical character reversed.

This is the syntax of the **REVERSE** function:

REVERSE Function

" **REVERSE**(*source_string*) "

Element	Description	Restrictions	Syntax
source_string	Expression that evaluates to a character string	Must be an expression, constant, column, or host variable of a type that can be converted to a character type	Expression on page 1003

The argument to the **REVERSE** function cannot have a user-defined data type. The built-in CHAR, LVARCHAR, NCHAR, NVARCHAR, and VARCHAR types are valid.

The **REVERSE** function returns a string of the same data type as its *source_string* argument.

If the expression that you specify as the argument evaluates to NULL, the return value is NULL.

For an argument that evaluates to string of `N` characters, the ordinal position `p` of each character in the *source_string* becomes `(N + 1 - p)` in the returned string. This inverts the sequence of characters from their original order in the *source_string*, so that the return value begins with the last character of the *source_string*, and ends with the first character of the *source_string*.

For example, the function expression `REVERSE('Mood')` returns the string `dooM` from the quoted-string argument. In both single-byte and multibyte code sets, only the ordinal positions are reversed, not the characters themselves. In the function expression above, 'd' does not become 'b', and each logical character in a multibyte code set (for example, **utf8**, or **GB2312-80**) is repositioned as a single logical unit.

If the argument evaluates to a single-character or to an empty *source_string*, the return value and the *source_string* are identical, as if the **REVERSE** function had no effect. For strings that include multiple characters, this equality is true only when the *source_string* is a palindrome. For character strings where `MOD(N,2) = 1`, the character in ordinal position `(N+1)/2` has the same middle position in both the *source_string* and in the returned string.

In the following example, the **REVERSE** function reverses a quoted string argument:

```
SELECT REVERSE('Able was I ere I saw Elba.') FROM Mirror_Table;
```

The following table shows the output of this SELECT statement.

(constant)
.ablE was I ere I saw elbA

REPLACE Function

The **REPLACE** function replaces specified characters within a source string with different characters.

REPLACE Function

" **REPLACE**(*source_string* ,*old_string* [,*new_string*]) "

Element	Description	Restrictions	Syntax
new_string	Character or characters that replace *old_string* in the return string	Must be an expression, constant, column, or host variable of a data type that can be converted to a character data type	Expression on page 1003
old_string	Character or characters in *source_string* that are to be replaced by *new_string*	Must be an expression, constant, column, or host variable of a data type that can be converted to a character data type	Expression on page 1003
source_string	String of characters argument to the **REPLACE** function	Must be an expression, constant, column, or host variable of a data type that can be converted to a character data type	Expression on page 1003

Any argument to the **REPLACE** function must be of a built-in data type.

The **REPLACE** function returns a copy of *source_string* in which every occurrence of *old_string* is replaced by *new_string*. If you omit the *new_string* option, every occurrence of *old_string* is omitted from the return string.

The return data type is its *source_string* argument. If a host variable is the source, the return value is either NVARCHAR or NCHAR, according to the length of the returned string, using the return type promotion rules that the section Return Types from the CONCAT Function on page 1115 describes.

In the following example, the **REPLACE** function replaces every occurrence of `xz` in the source string with `t`:

```
SELECT REPLACE('Mighxzy xzime', 'xz', 't')
   FROM mytable;
```

The following table shows the output of this SELECT statement.

(constant)
`Mighty time`

LPAD Function

The **LPAD** function returns a copy of *source_string* that is left-padded to the total number of characters specified by *length*.

LPAD Function

" **LPAD** (*source_string* , *length* [, *pad_string*]) "

Element	Description	Restrictions	Syntax
length	Integer value that specifies total number of characters in the returned string	Must be an expression, constant, column, or host variable of a data type that can be converted to an integer data type	Literal Number on page 1198
pad_string	String that specifies the pad character or characters	Must be an expression, constant, column, or host variable of a data type that can be converted to a character data type	Expression on page 1003
source_string	String that serves as input to the LPAD function	Must be an expression, constant, column, or host variable of a data type that can be converted to a character data type	Expression on page 1003

Any argument to the **LPAD** function must be of a built-in data type.

The *pad_string* parameter specifies the character or characters to be used for padding the source string. The sequence of pad characters occurs as many times as necessary to make the return string the storage length specified by *length*.

The series of pad characters in *pad_string* is truncated if it is too long to fit into *length*. If you specify no *pad_string*, the default value is a single blank (ASCII 32) character.

The return data type is based on the three arguments, using the return type promotion rules that the section Return Types from the CONCAT Function on page 1115 describes.

In the following example, the user specifies that the source string is to be left-padded to a total length of 16 characters. The user also specifies that the pad characters are a series consisting of a hyphen and an underscore (-_).

```
SELECT LPAD('Here we are', 16, '-_') FROM mytable;
```

The following table shows the output of this SELECT statement.

(constant)
-_-_-Here we are

RPAD Function

The **RPAD** function returns a copy of *source_string* that is right-padded to the total number of characters that the *length* argument specifies.

RPAD Function

" **RPAD** (*source_string* , *length* [, *pad_string*]) "

Element	Description	Restrictions	Syntax
length	The total number of characters in the returned string	Must be an expression, constant, column, or host variable that returns an integer	Literal Number on page 1198
pad_string	String that specifies the pad character or characters	Must be an expression, column, constant, or host variable of a data type that can be converted to a character data type	Expression on page 1003
source_string	String that serves as input to the RPAD function	Same as for *pad_string*	Expression on page 1003

Any argument to the **RPAD** function must be of a built-in data type.

The *pad_string* parameter specifies the pad character or characters to be used to pad the source string.

The series of pad characters occurs as many times as necessary to make the return string reach the length that *length* specifies. The series of pad characters in *pad_string* is truncated if it is too long to fit into *length*. If you omit the *pad_string* parameter, the default value is a single blank space (the ASCII 32 character).

The return data type is based on the *source_string* and *pad_string* arguments, if both are specified. If a host variable is the source, the return value is either NVARCHAR or NCHAR, according to the length of the returned string, using the return type promotion rules that the section Return Types from the CONCAT Function on page 1115 describes.

The UNLOAD feature of truncates trailing blanks in CHAR or NCHAR columns, even if the **RPAD** function has appended blank characters to the data value. You must explicitly cast the CHAR or NCHAR value to a VARCHAR, LVARCHAR, or NVARCHAR data type if you need UNLOAD to preserve trailing blank characters or nonprintable characters in a value that **RPAD** returns.

In the following example, the user specifies that the source string is to be right-padded to a total length of 18 characters. The user also specifies that the pad characters to be used are a sequence consisting of a question mark and an exclamation point (?!)

```
SELECT RPAD('Where are you', 18, '?!')
   FROM mytable;
```

The following table shows the output of this SELECT statement.

(constant)
Where are you?!?!?

CHR Function

This function accepts an unsigned integer argument, and returns a single logical character.

The **CHR** function has this syntax:

CHR Function

"`CHR(`*expression*`)`"

Element	Description	Restrictions	Syntax
expression	Expression that evaluates to a nonnegative whole number less than 256	Must be an integer in the range `0` through `255` (inclusive)	Expression on page 1003

The data type of the return value is VARCHAR(1).

The argument can be SMALLINT, INTEGER, SERIAL, INT8, SERIAL8, BIGINT, or BIGSERIAL. The argument must evaluate to a whole number in the range `0` through `255`.

If the argument is an integer in the range `0` through `127`, the return value is the corresponding single-byte ASCII code point. For a listing of the characters corresponding to the ASCII code points `0` through `127`, see #unique_1071.

If the argument is an integer in the range `128` through `255`, the return value is the corresponding 2-byte code point in the default code set.

- On UNIX™ platforms, the default code set is ISO8859-1.
- On Windows™ platforms, the default code set is Microsoft™ 1252.

Case-Conversion Functions

The case-conversion functions perform lettercase conversion on alphabetic characters. In the default locale, only the ASCII characters in the ranges `A - Z` and `a - z` can be modified by these functions, which enable you to perform case-insensitive searches in your queries and to specify the format of the output.

The case-conversion functions are **UPPER**, **LOWER**, and **INITCAP**. The following diagram shows the syntax of these case-conversion functions.

Case-Conversion Functions

"`{ UPPER | LOWER | INITCAP }`"

"`(`*expression*`)`"

Element	Description	Restrictions	Syntax
expression	Expression returning a character string	Must be a built-in character type. If a host variable, its length must be long enough to store the converted string.	Expression on page 1003

The *expression* must return a character data type. When a column expression is specified, the column data type returned by the database server is that of *expression*. For example, if the input type is CHAR, the output type is also CHAR.

Argument to these functions must be of the built-in data types.

In all locales, the byte length returned from the description of a column with a case-conversion function is the input byte length of the source string. If you use a case-conversion function with a multibyte *expression* argument, the conversion might increase or decrease the length of the string. If the byte length of the result string exceeds the byte length *expression*, the database server truncates the result string to fit into the byte length of *expression*.

Only characters designated as ALPHA class in the locale file are converted, and this occurs only if the locale recognizes the construct of lettercase.

If *expression* evaluates to NULL, the result of a case-conversion function is also NULL.

The database server treats a case-conversion function as an SPL routine in the following instances:

- If it has no argument
- If it has one argument, and that argument is a named argument
- If it has more than one argument
- If it appears in the Projection list with a host variable as an argument

If none of the conditions in the preceding list are met, the database server treats a case-conversion function as a system function.

The following example uses all the case-conversion functions in the same query to specify multiple output formats for the same value:

```
Input value:

SAN Jose

Query:

SELECT City, LOWER(City), LOWER("City"),
   UPPER (City), INITCAP(City)
      FROM Weather;

Query output:

SAN Jose   san jose   city   SAN JOSE   San Jose
```

UPPER Function

The **UPPER** function accepts an *expression* argument and returns a character string in which every lowercase alphabetical character in the *expression* is replaced by a corresponding uppercase alphabetic character.

The following example uses the **UPPER** function to perform a case-insensitive search on the **lname** column for all employees with the last name of **Curran**:

```
SELECT title, INITCAP(fname), INITCAP(lname) FROM employees
   WHERE UPPER (lname) = "CURRAN"
```

Because the **INITCAP** function is specified in the projection list, the database server returns the results in a mixed-case format. For example, the output of one matching row might read: `accountant James Curran.`

LOWER Function

The **LOWER** function accepts an *expression* argument and returns a character string in which every uppercase alphabetic character in the *expression* is replaced by a corresponding lowercase alphabetic character.

The following example shows how to use the **LOWER** function to perform a case-insensitive search on the **City** column. This statement directs the database server to replace all instances (that is, any variation) of the words `san jose`, with the mixed-case format, `San Jose`.

```
UPDATE Weather SET City = "San Jose"
   WHERE LOWER (City) = "san jose";
```

INITCAP Function

The **INITCAP** function returns a copy of the *expression* in which every word in the *expression* begins with an uppercase letter. With this function, a *word* begins after any character other than a letter. Thus, in addition to a blank space, symbols such as commas, periods, colons, and so on, introduce a new word.

For an example of the **INITCAP** function, see UPPER Function on page 1132.

Case-conversion functions in NLSCASE INSENSITIVE databases

The **UPPER** and **LOWER**, case-conversion functions were designed to support case-insensitive queries in a case-sensitive database. They are less often needed in databases that have the NLSCASE INSENSITIVE attribute, because the NCHAR and NVARCHAR data types can support case-insensitive queries without calling these functions. You can invoke the case-conversion functions in NLSCASE INSENSITIVE databases, where their effects on CHAR, LVARCHAR, and VARCHAR data types are the same as in case-sensitive databases.

In a database created with the NLSCASE INSENSITIVE option, the database server disregards the lettercase of NCHAR and NVARCHAR values. Expressions that call case-conversion functions can return different results from what a case-sensitive database would return, if the expression references NCHAR or NVARCHAR objects, or if the database server evaluates the expression with an explicit or implicit cast to an NCHAR or NVARCHAR data type.

When the **UPPER** , **LOWER**, or **INITCAP** function is used in evaluating a string expression in a database with the NLSCASE INSENSITIVE property, the database server invokes the function, and applies to its return value the data-type promotion rules that are summarized in the topic Return Types from CONCAT and String Functions on page 1115.

- If the expression evaluates to a CHAR, LVARCHAR, or VARCHAR data type, the database server can use that result in case-sensitive operations, if those operations do not involve NCHAR or NVARCHAR objects.
- If the expression evaluates to an NCHAR or NVARCHAR value after the **UPPER**, **LOWER**, or **INITCAP** function has executed, the case of letters in this result is disregarded in subsequent operations that use this return value from the expression.

The following example illustrates this behavior in an NLSCASE INSENSITIVE database, in which table **t1** has a character column of each of the five built-in character data types. The table stores three rows, in which each column stores the same lettercase variants of a 3-letter character string:

```
CREATE DATABASE db NLSCASE INSENSITIVE;
CREATE TABLE t1 (
   c1 NCHAR(20),
   c2 NVARCHAR(20),
   c3 CHAR(20),
   c4 VARCHAR(20),
   c5 LVARCHAR(20)) ;
INSERT INTO t1 values ('ibm', 'ibm', 'ibm', 'ibm', 'ibm');
INSERT INTO t1 values ('Ibm', 'Ibm', 'Ibm', 'Ibm', 'Ibm');
INSERT INTO t1 values ('IBM', 'IBM', 'IBM', 'IBM', 'IBM');
```

In the following example, the database server applies the **UPPER** function to the NCHAR column **c1** and then applies a case-insensitive rule to return all values in that column that match the `'IBM'` string constant.

```
SELECT c1 FROM t1 WHERE UPPER(c1) = 'IBM';
```

Because NCHAR values are not case sensitive in this database, the query returns the column **c1** value from every row in the table, because in each row the sequence of letters matches the string constant, using a case-insensitive rule that ignores the letter case of the column values:

```
c1

ibm
Ibm
IBM
```

The same result set (namely `'ibm'`, `'Ibm'`, and `'IBM'`) would be also returned by the following modifications to the same query on the same table:

- If the projection clause specified any other column, rather than **c1**, because every column stores the same values, and the NCHAR value that **UPPER** returns makes the WHERE clause true for all lettercase variants of the string `'IBM'` in this database.
- If the `'IBM'` string in the WHERE clause were any other lettercase variant of the same sequence of letters, because NCHAR data types are not processed by case-sensitive rules in this database.

- If the NVARCHAR column **c2**, rather than NCHAR column **c1**, were the argument to the case-conversion function, because both NCHAR or NVARCHAR are case-insensitive data types in this database.
- If the case-conversion function **LOWER** or **INITCAP**, rather than **UPPER**, were applied to column **c1**, because every (case-variant) value that NCHAR column matches `'IBM'` in this database.
- If no case-conversion function were called, but the WHERE condition instead specified `c1 = 'IBM'`, because case-conversion functions have no effect as query filters on NCHAR or NVARCHAR arguments in this NLSCASE INSENSITIVE database.

Substring functions

The built-in SQL substring functions return substrings from character string arguments, or return positional information for operations on substrings.

Substring Functions

(explicit id unique_1628_Connect_42_sbfu001) unique_1628_Connect_42_sbfu001 (explicit id unique_1628_Connect_42_sbfu002) unique_1628_Connect_42_sbfu002 (explicit id unique_1628_Connect_42_sbfu003) unique_1628_Connect_42_sbfu003 (explicit id unique_1628_Connect_42_sbfu004) unique_1628_Connect_42_sbfu004 (explicit id unique_1628_Connect_42_sbfu010) unique_1628_Connect_42_sbfu010 (explicit id unique_1628_Connect_42_sbfu005) unique_1628_Connect_42_sbfu005 (explicit id unique_1628_Connect_42_sbfu007) unique_1628_Connect_42_sbfu007 (explicit id unique_1628_Connect_42_sbfu008) unique_1628_Connect_42_sbfu008 (explicit id unique_1628_Connect_42_sbfu009) unique_1628_Connect_42_sbfu009

```
" { [ <CHARINDEX function> ] | [ <INSTR function> ] | <LEFT function> [] | <RIGHT function> [] | <SUBSTR function> [] |
<SUBSTRB function> [] | <SUBSTRING function> [] | <SUBSTRING_INDEX function> [] } "
```

Sections that follow describe the syntax and usage of these substring functions.

CHARINDEX function

The **CHARINDEX** function searches a character string for the first occurrence of a target substring, where the search begins at a specified or default character position within the source string.

The **CHARINDEX** function has this syntax:

CHARINDEX function

```
" CHARINDEX( substring, source [ ,start_position ] ) "
```

Element	Description	Restrictions	Syntax
source_ string	Expression that evaluates to a character string	Must be an expression, constant, column, or host variable of a built-in character type, or a type that can be converted to a character type	Expression on page 1003
start_ position	Ordinal position to begin the search in *source*, where 1 is the first logical character	Must be an expression, constant, column, or host variable of a built-in integer type, or that can be converted to an integer	Expression on page 1003
substring	Expression that evaluates to a character string	Must be an expression, constant, column, or host variable of a built-in character data type, or that can be converted to a character type	Expression on page 1003

Arguments to **CHARINDEX** cannot be user-defined data types.

If either *source* or the *substring* is NULL, this function returns NULL.

If the optional *start_position* value is less than 1, or if you omit this argument, none, the search for *substring* begins at the first logical character in the *source*, as if you had specified `1` as the starting position.

If no expression matching the *substring* is found, **CHARINDEX** returns zero (`0`). Otherwise it returns the ordinal position of the first logical character in the first occurrence of *substring*.

If you specify a *start_position* greater than 1, any *substring* that begins before *start_position* is ignored, and the function returns one of the following values:

- either the position of the first logical character in the first matching substring whose ordinal position is equal to or greater than *start_position*,
- or else zero (`0`), if no occurrence of *substring* in *source* begins at or follows *start_position*, or if *start_position* is greater than the number of logical characters in *source*.

In locales that support multibyte character sets, the return value is the ordinal value among logical characters in the *source*. In single-byte locales, such as the default locale, the return value is equivalent to the byte position, where the first byte is in position `1`.

In databases created with the NLSCASE INSENSITIVE option, if either *source* or *substring* is an NCHAR or NVARCHAR data type, the database server ignores variants in letter case in determining whether a given substring of *source* matches the target *substring*.

The following function expression returns `9`:

```
CHARINDEX('com','www.ibm.com')
```

In the example above, **CHARINDEX** begins its search at the default starting position of 1.

The following function expression returns `2`:

```
CHARINDEX('w','www.ibm.com',2)
```

In the example above, because the last argument begins the search at position `2`, **CHARINDEX** ignores two other matching substrings:

- `'w'` in position `1`, because the search begins at `2`,
- and `'w'` in position `3`, because the function returns the position of only the first occurrence of a matching substring.

INSTR function

The **INSTR** function searches a character string for a specified substring, and returns the character position in that string where an occurrence of that a substring ends, based on a count of substring occurrences.

The **INSTR** function has this syntax:

INSTR function

> " `INSTR(`*source_string*, *substring* [,*start* [,*count*]] `)` "

Element	Description	Restrictions	Syntax
count	Expression that evaluates to an integer `> 0`	Must be an expression, constant, column, or host variable of a built-in integer type, or that can be converted to an integer.	Expression on page 1003
source_ string	Expression that evaluates to a character string	Must be an expression, constant, column, or host variable of a built-in character data type, or that can be converted to a character type	Expression on page 1003
start	Ordinal position to begin the search in *source_string*, where 1 is the first logical character	Must be an expression, constant, column, or host variable of a built-in integer type, or that can be converted to a positive or negative integer	Expression on page 1003
substring	Expression that evaluates to a character string	Must be an expression, constant, column, or host variable of a built-in character data type, or that can be converted to a character type	Expression on page 1003

Arguments to **INSTR** cannot be user-defined data types.

The function returns NULL in each of these cases:

- *count* is less than or equal to zero (`0`).
- *source_string* is NULL or of zero length.
- *substring* is NULL or of zero length.

The return value is zero (`0`) in each of the following cases:

- if no occurrences of *substring* are found in *source_string*,
- if *start* is greater than the length of *source_string*.
- If fewer than *count* occurrences of *substring* are in the *source_string*,

If you omit the optional *count* argument, the default *count* value is 1.

In locales that support multibyte character sets, the return value is the ordinal value among logical characters in the *source_string*. In single-byte locales, such as the default locale, the return value is equivalent to the byte position, where the first byte is in position `1`.

The *start* position

If *start* is omitted or is specified as zero, the search for *substring* begins at character position 1. If *start* is negative, the search for occurrences of *substring* begins at the end of *source_string*, and proceeds towards the beginning.

- In a left-to-right locale, a negative *start* value specifies a right-to-left search.
- In a right-to-left locale, a negative *start* value specifies a left-to-right search.

In both types of locales, however, the search begins at the logical character position within *source_string* that corresponds to the absolute value of *start*.

In a right-to-left locale, a negative *start* value specifies a left-to-right search.

Examples of INSTR function expressions

The following expressions are all based on the same *source_string* and *substring*. This example returns `3`, as the character position of the first `'er'` substring:

```
INSTR("wwerw.ibm.cerom", "er")
```

In the example above, both *start* and *count* default to `1`.

The nest example starts the search at the 2nd character position, with a default *count* of `1`:

```
INSTR("wwerw.ibm.cerom", "er", 2)
```

The expression above returns `3`, the position of the first character in the first `'er'` substring that a left-to-right search encounters.

The next example specifies a count of `2`, starting the search in the first character of the *source_string*:

```
INSTR("wwerw.ibm.cerom", "er", 1, 2)
```

The expression above returns `12`, the character position where the second `'er'` begins.

The following example specifies `-5` as the starting position, and the *count* specifies the first occurrence of `"er"` between the 5th position and the beginning of the *source_string*:

```
INSTR("wwerw.ibm.cerom", "er", -5, 1)
```

This returns `3`, corresponding to the occurrence of the `"er"` substring that begins in that position. The negative *start* argument specifies a right-to-left search, but the return value is `3`, because the reading direction of strings and substrings in the default locale is left-to-right.

LEFT function

The **LEFT** function returns a substring consisting of the leftmost *N* characters from a string argument.

The function has this syntax:

LEFT function

"**LEFT**(*source_string*, *position*)"

Element	Description	Restrictions	Syntax
position	Ordinal position (from the left) in the string; this character and all to the left are to be returned	Must be an expression, constant, column, or host variable of a built-in integer type, or that can be converted to an integer	Expression on page 1003
source_string	Expression that evaluates to a character string	Must be an expression, constant, column, or host variable of a data type that can be converted to a character type	Expression on page 1003

The arguments to the **LEFT** function cannot be user-defined data types.

What the **LEFT** function returns depends on the number of logical characters in *source_string* and on the value of *position*:

- If *source_string* evaluates to a string with more than *position* characters, the return value is a substring of *source_string*, consisting of all characters to the left of the specified *position*.
- If *source_string* evaluates to a string with no more than *position* characters, the return value is the entire *source_string*.
- If *source_string* evaluates to NULL, or if *position* is zero or negative, then NULL is returned.
- If no *position* argument is specified, no string value is returned, and an exception is issued.

The return data type is the same as its *source_string* argument. If a host variable is the source, the return value is either NVARCHAR or NCHAR, according to the length of the returned string, using the return type promotion rules that the section Return Types from the CONCAT Function on page 1115 describes.

The following function expression requests the first five characters of a quoted string:

```
LEFT('www.ibm.com',5)
```

In this example, the **LEFT** function returns the substring `www.i`

RIGHT function

The **RIGHT** function returns a substring consisting of the rightmost *N* characters from a string argument.

The function has this syntax:

RIGHT function

"`RIGHT(`*source_string*`,` *position*`)`"

Element	Description	Restrictions	Syntax
position	Ordinal position (from the right) in the string; this character and all to the right are to be returned	Must be an expression, constant, column, or host variable of a built-in integer type, or that can be converted to an integer	Expression on page 1003
source_string	Expression that evaluates to a character string	Must be an expression, constant, column, or host variable of a data type that can be converted to a character type	Expression on page 1003

The arguments to the **RIGHT** function cannot be user-defined data types.

In left-to-right locales, such as the default U.S. English locale, this function returns a substring of trailing characters from the *source_string*.

What the **RIGHT** function returns depends on the number of logical characters in *source_string* and on the value of *position*:

- If *source_string* evaluates to a string with more than *position* characters, the return value is a substring of *source_string*, consisting of all characters to the right of the specified *position*.
- If *source_string* evaluates to a string with no more than *position* characters, the return value is the entire *source_string*.
- If *source_string* evaluates to NULL, or if *position* is zero or negative, then NULL is returned.
- If no *position* argument is specified, no string value is returned, and an exception is issued.

The return data type is the same as its *source_string* argument. If a host variable is the source, the return value is either NVARCHAR or NCHAR, according to the length of the returned string, using the return type promotion rules that the section Return Types from the CONCAT Function on page 1115 describes.

The following function expression requests the last five characters of a quoted string:

```
RIGHT('www.ibm.com',5)
```

In this example, the **RIGHT** function returns the substring `m.com`

SUBSTR function

The **SUBSTR** function has the same purpose as the **SUBSTRING** function (to return a subset of a source string), but it uses different syntax.

SUBSTR Function

" **SUBSTR**(*source_string* , *start_position* [, *length*]) "

Element	Description	Restrictions	Syntax
length	Number of characters to be returned from source_string	Must be an expression, literal, column, or host variable that returns an integer	Expression on page 1003
source_string	String that serves as input to the **SUBSTR** function	Must be an expression, literal, column, or host variable of a data type that can be converted to a character data type	Expression on page 1003
start_position	Column position in *source_string* where the **SUBSTR** function starts to return characters	Must be an integer expression, literal, column, or host variable. Can have a plus sign (+), a minus sign (-), or no sign.	Literal Number on page 1198

Any argument to the **SUBSTR** function must be of a built-in data type.

The **SUBSTR** function returns a subset of *source_string*. The subset begins at the column position that *start_position* specifies. The following table shows how the database server determines the starting position of the returned subset based on the input value of the *start_position.*

Value of Start_Position	How the Database Server Determines the Starting Position of the Returned Subset
Positive	Counts forward from the first character in *source_string*
Zero (0)	Counts forward from the first character in *source_string* (that is, treats a *start_position* of 0 as equivalent to 1)
Negative	Counts backward from an origin that immediately follows the last character in *source_string*. A value of -1 returns the last character in *source_string*.

The *length* parameter specifies the number of logical characters (not the number of bytes) in the subset. If you omit the *length* parameter, the **SUBSTR** function returns the entire portion of *source_string* that begins at *start_position.*

If you specify a negative *start_position* whose absolute value is greater than the number of characters in *source_string*, or if *length* is greater than the number of characters from *start_position* to the end of *source_string*, **SUBSTR** returns NULL. (In this case, the behavior of **SUBSTR** is different from that of the **SUBSTRING** function, which returns all the characters from *start_position* to the last character of *source_string*, rather than returning NULL.)

The return data type is that of the *source_string* argument. If a host variable is the source, the return value is either NVARCHAR or NCHAR, according to the length of the returned string, using the return type promotion rules that the section Return Types from the CONCAT Function on page 1115 describes.

The following example specifies that the string of characters to be returned begins at a starting position 3 characters before the end of a 7-character *source_string*. This implies that the starting position is the fifth character of *source_string*. Because the user does not specify a value for *length*, the database server returns a string that includes all characters from character-position 5 to the end of *source_string*.

```
SELECT SUBSTR('ABCDEFG', -3)
   FROM mytable;
```

The following table shows the output of this SELECT statement.

(constant)
EFG

SUBSTRB function

Returns a substring of a string, beginning at a specified position in the string.

SUBSTRB Function

" **SUBSTRB**(*source_string* , *starting_position* [, *length*] "

Element	Description
length	An expression that specifies the length of the result in bytes. If specified, the expression must return a value that is a built-in numeric, CHAR, or VARCHAR data type. If the value is not of type INTEGER, it is implicitly cast to INTEGER before the function is evaluated. If the value of *length* is greater than the number of bytes from the starting position to the end of the string, the length of the result is equal to is the length of the first argument minus the starting position, plus one. If the value of *length* is less than or equal to zero, the result of SUBSTRB is a NULL string. The default for *length* is the number of bytes from the position specified by *starting_position* to the last byte of the string. When *length* is specified, the length of the result string is truncated to the value of *length*. In the following example, where *my_string* is a 10-byte string, the result string is limited to 5 bytes: `substrB(my_string, 3, 5)`

Element	Description
	If, in the preceding example, *my_string* is a 4-byte string and the starting position is the third byte, a 2-byte string is returned. If *length* is not specified, the length of the result is the length of *source_string* beginning from the *starting_position*. In the following example, where *my_string* is a 10-byte string, an 8-byte string is returned: `substrB(my_string, 3)`
source_string	An expression that specifies the string from which the result is derived. The expression must return a value that is a built-in string, numeric, or datetime data type. If the value is not a string data type, it is implicitly cast to NVARCHAR before the function is evaluated. A NULL value is returned for a zero length result.
starting_position	An expression that specifies the starting position in string of the beginning of the result substring. The expression must return a value that is a built-in numeric, CHAR, or VARCHAR data type. If the value is not of type INTEGER, it is implicitly cast to an INTEGER before the function is evaluated. If *starting_position* is positive, then the starting position is calculated from the beginning of the string. If *starting_position* is greater than the *length* of string, then a null string is returned. If *starting_position* is negative, then the starting position is calculated from the end of the string and by counting backwards the number of bytes. If the absolute value of *starting_position* is greater than the *length* of *source_string*, then a null string is returned. If *starting_position* is 0, then a starting position of 1 is used. **Note:** All units of *length* and *starting_position* are expressed in terms of bytes, even for strings encoded in multibyte code sets. SUBSTR uses the logical character size for multibyte strings. For example, if *starting_position* is `2` in the legacy SUBSTR and the first character of a multibyte string requires `3` bytes of storage, the `2` represents the fourth byte in the string. In SUBSTRB, the `2` represents the second byte in the string.

If *source_string* is a CHAR or VARCHAR data type, the result of the function is a VARCHAR data type. Informix® does not support multiple code pages; instead, Informix® JDBC or ODBC translates the code page to the database.

If any argument is null, the result is the null value.

In dynamic SQL, *source_string*, *starting_position*, and *length* can be represented by a host variable. If a host variable is used for *source_string*, the data type of the operand is VARCHAR, and the operand can be nullable.

Though not explicitly stated in the result definitions above, the semantics imply that if *source_string* is a multi-byte character string, the result might contain fragments of multi-byte characters, depending on the values of *starting_position* and *length*. For example, the result could possibly begin with the second byte of a multi-byte character, or end with the first byte of a multi-byte character. The SUBSTRB function detects these partial characters and replaces each byte of an incomplete

character with a single blank character. SUBSTRB returns a fixed number of bytes; with SUBSTR, the number of returned varies according to the multibyte string.

SUBSTRING function

The **SUBSTRING** function returns a subset of a character string.

SUBSTRING Function

" **SUBSTRING**(*source_string* **FROM***start_position* [**FOR***length*]) "

Element	Description	Restrictions	Syntax
length	Number of characters to return from *source_string*	Must be an expression, constant, column, or host variable that returns an integer	Literal Number on page 1198
source_string	String argument to the **SUBSTRING** function	Must be an expression, constant, column, or host variable whose value can be converted to a character data type	Expression on page 1003
start_position	Position in *source_string* of first returned character	Must be an expression, constant, column, or host variable that returns an integer	Literal Number on page 1198

Any argument to the **SUBSTRING** function must be of a built-in data type.

The return data type is that of the *source_string* argument. If a host variable is the source, the return value is either NVARCHAR or NCHAR, according to the length of the returned string, using the return type promotion rules that the section Return Types from the CONCAT Function on page 1115 describes.

The subset begins at the column position that *start_position* specifies. The following table shows how the database server determines the starting position of the returned subset based on the input value of the *start_position.*

Value of Start_Position	How the Database Server Determines the Starting Position of the Return Subset
Positive	Counts forward from the first character in *source_string* For example, if *start_position* = 1, the first character in the *source_string* is the first character in the returned subset.
Zero (0)	Counts from one position before (that is, to the left of) the first character in *source_string* For example, if *start_position* = 0 and *length* = 1, the database server returns NULL, whereas if *length* = 2, the database server returns the first character in *source_string*.
Negative	Counts backward from one position after (that is, to the right of) the last character in *source_string*

Value of Start_Position	How the Database Server Determines the Starting Position of the Return Subset
	For example, if *start_position* = -1, the starting position of the returned subset is the last character in *source_string.*

The size of the subset is specified by *length*. The *length* parameter refers to the number of logical characters, rather than to the number of bytes. If you omit the *length* parameter, or if you specify a *length* that is greater than the number of characters from *start_position* to the end of *source_string*, the **SUBSTRING** function returns the entire portion of *source_ string* that begins at *start_position.* The following example specifies that the subset of the source string that begins in column position `3` and is two characters long should be returned:

```
SELECT SUBSTRING('ABCDEFG' FROM 3 FOR 2) FROM mytable;
```

The following table shows the output of this SELECT statement.

(constant)
CD

In the following example, the user specifies a negative *start_position* for the return subset:

```
SELECT SUBSTRING('ABCDEFG' FROM -3 FOR 7)
   FROM mytable;
```

The database server starts at the `-3` position (four positions before the first character) and counts forward for `7` characters. The following table shows the output of this SELECT statement.

(constant)
ABC

SUBSTRING_INDEX function

The **SUBSTRING_INDEX** function searches a character string for a specified delimiter character, and returns a substring of the leading or trailing characters, based on a count of a delimiter that you specify as an argument to the function.

The **SUBSTRING_INDEX** function has this syntax:

SUBSTRING_INDEX function

" **SUBSTRING_INDEX**(*source_string* , *delimiter* , *count*) "

Element	Description	Restrictions	Syntax
source_ string	Expression that evaluates to a character string	Must be an expression, constant, column, or host variable of a built-in character data type, or that can be converted to a character type	Expression on page 1003

Element	Description	Restrictions	Syntax
count	Expression that evaluates to a positive or negative integer	Must be an expression, constant, column, or host variable of a built-in integer type, or that can be converted to an integer.	Expression on page 1003
delimiter	Expression that evaluates to a character string	Must be an expression, constant, column, or host variable of a built-in character data type, or that can be converted to a character type	Expression on page 1003

Arguments to **SUBSTRING_INDEX** cannot be user-defined data types.

This function returns NULL in each of the following cases:

- *source_string* is NULL
- *delimiter* is NULL
- *count* = zero (`0`).

If the search finds fewer than *count* delimiters in the *source_string*, the return value is the entire *source_string*.

The return value has the same data type as the *source_string*.

For *source_string*, the sign of *count* determines whether the returned value is a substring of the leading characters or of the trailing characters in *source_string*:

- The last character in the returned substring immediately precedes the Nth occurrence of that delimiter in a substring of leading characters, for N = *count*.

  For example, the function expression

  ```
  SUBSTRING_INDEX("www.ibm.com", ".", 2)
  ```

  returns the leading characters `www.ibm` because *count* `> 0`.
- The first character in the returned substring immediately precedes the Nth occurrence of that delimiter in a substring of trailing characters, for N = *count* `< 0`.

  For example, the function expression

  ```
  SUBSTRING_INDEX("www.ibm.com", ".", -2)
  ```

  returns the trailing characters `ibm.com` because *count* `< 0`.

The examples above apply to left-to-right locales, such as the default U.S. English locale, in which a negative value of *count* causes this function to return a substring of trailing characters from the *source_string*, and a positive value of *count* causes this function to return a substring of leading characters from the *source_string*,

In locales that support multibyte character sets, the return value is the ordinal value among logical characters in the *source_string*. In single-byte locales, such as the default locale, the return value is equivalent to the byte position, where the first byte is in position `1`.

FORMAT_UNITS Function

The **FORMAT_UNITS** function can interpret strings that specify a number and the abbreviated names of units of memory or of mass storage.

This built-in function can accept one, two, or three quoted string arguments. You can invoke **FORMAT_UNITS** in SQL statements that process size specifications expressed by standard abbreviations for bytes or for larger units (such as kilobytes, megabytes, gigabytes, and so forth) of memory or of mass storage.

The **FORMAT_UNITS** function can also be called internally in the **sysadmin** database by the SQL administration API **ADMIN** and **TASK** functions, which are described in the *HCL® Informix® Administrator's Reference*.

FORMAT_UNITS Function

" **FORMAT_UNITS**({ '*numberunits* ' | [{ '*number*' | *number* } , '*units*' [, { '*precision*' | *precision* }]] }) "

Element	Description	Restrictions	Syntax
number	Expression that evaluates to the number of storage or memory *units*	Must be a literal number or a quoted string specifying a number that can be converted to FLOAT	Expression on page 1003
precision	Integer number of significant digits to return from *number*	Must be a literal number or a quoted string specifying an integer	Expression on page 1003
units	Abbreviation of a unit of storage or memory; the default is 'B' (for bytes)	Must begin with 'B', 'K', 'M', 'G', 'T', 'P', or 'PB' (or the lowercase forms of these letters). Any trailing characters are ignored.	Quoted String on page 1202

This built-in function can accept one, two, or three arguments. The returned value is a character string that shows the specified *number* and an appropriate format label that shows the storage units. If you specify a *precision* as the last argument, the *number* is returned with that precision. Otherwise, the *number* is formatted to precision 3 (`%3.3lf`) by default.

The same notation also applies to arguments to all SQL administration API **ADMIN** and **TASK** commands (except for commands that emulate the Enterprise Replication **cdr** utility) that specify sizes of memory, of disk storage, or of address offsets:

Notation

Corresponding Units

'B' or 'b'

Bytes (= 2 to the power 0)

'K' or 'k'

Kilobytes (= 2 to the power 10)

'M' or 'm'

Megabytes (= 2 to the power 20)

'G' or 'g'

Gigabytes (= 2 to the power 30)

'T' or 't'

Terabytes (= 2 to the power 40)

'PB'

Petabytes (= 2 to the power 50)

'P'

Pages (= 2 kilobytes or 4 kilobytes, depending on the base page size of the system)

The initial letter in the *unit* specification (`'B'`, `'K'`, `'M'`, `'G'` or `'T'`) determines the units of measure, and any trailing characters are ignored. An exception, however, is if the initial letter `'P'` (or `'p'`) is immediately followed by 'B' or 'b' in the string, because in this case the string is interpreted as petabytes. Any other string starting with "P" (such as "`PA`", "`pc`", "`PhD`", "`papyrus`" and so forth) is interpreted as specifying *pages*, rather than *petabytes*.

If one argument provides both the *number* and *units* specifications, Informix® ignores any whitespace that separates the *number* specification from the *units* specification within the same argument to the **FORMAT_UNITS**, or SQL administration API **ADMIN** or **TASK** functions. For example, the specifications `'128M'` and `'128 m'` are both interpreted as 128 megabytes.

The following examples invoke the **FORMAT_UNITS** function with a single argument:

```
EXECUTE FUNCTION FORMAT_UNITS('1024 M');
```

The following character string value is returned.

(expression)
1.00 GB

```
SELECT FORMAT_UNITS('1024 k') FROM systables WHERE tabid=1;
```

The following character string value is returned.

(expression)
1.00 MB

```
SELECT FORMAT_UNITS(tabid || 'M') FROM systables WHERE tabid=100;
```

The following character string value is returned.

(expression)
100 MB

The following examples show calls to the **FORMAT_UNITS** function with two arguments:

```
EXECUTE FUNCTION FORMAT_UNITS(1024, 'k');
```

The following character string value is returned.

(expression)
1.00 MB

```
SELECT FORMAT_UNITS( SUM(chksize), 'P') SIZE,
   FORMAT_UNITS( SUM(nfree), 'p') FREE FROM syschunks;

size  117 MB
free  8.05 MB
```

This query returns the string values `size 117 MB` and `free 8.05 MB`.

The following examples show calls to the **FORMAT_UNITS** function with three arguments:

```
EXECUTE FUNCTION FORMAT_UNITS(1024, 'k', 4);
```

The following character string value is returned.

(expression)
1.000 MB

```
SELECT FORMAT_UNITS( SUM(chksize), 'P', 4), SIZE,
   FORMAT_UNITS( SUM(nfree), 'p', 4) FREE FROM syschunks;

size  117.2 MB
free  8.049 MB
```

This query returns the string values `size 117.2 MB` and `free 8.047 MB`. These results differ from the previous example of a query only in their non-default precision, which the last argument to **FORMAT_UNITS** specifies.

IFX_ALLOW_NEWLINE Function

The **IFX_ALLOW_NEWLINE** function sets a newline mode that allows newline characters in quoted strings or disallows newline characters in quoted strings within the current session.

The **IFX_ALLOW_NEWLINE** function has the following syntax.



If you enter `'t'` as the argument of this function, you enable newline characters in quoted strings in the session. If you enter `'f'` as the argument, you disallow newline characters in quoted strings in the session.

You can set the newline mode for all sessions by setting the ALLOW_NEWLINE parameter in the `ONCONFIG` file to a value of `0` (newline characters not allowed) or to a value of `1` (newline characters allowed). If you do not set this configuration parameter, the default value is `0`. Each time you start a session, the new session inherits the newline mode set in the `ONCONFIG` file. To change the newline mode for the session, execute the **IFX_ALLOW_NEWLINE** function. Once you have set the newline mode for a session, the mode remains in effect until the end of the session or until you execute the **IFX_ALLOW_NEWLINE** function again within the session.

In the following example, assume that you did not specify any value for the ALLOW_NEWLINE parameter in the `ONCONFIG` file, so, by default, newline characters are not allowed in quoted strings in any session. After you start a new session, you can enable newline characters in quoted strings in that session by executing the **IFX_ALLOW_NEWLINE** function:

```
EXECUTE PROCEDURE IFX_ALLOW_NEWLINE('t');
```

In ESQL/C, the newline mode that is set by the ALLOW_NEWLINE parameter in the `ONCONFIG` file or by the execution of the **IFX_ALLOW_NEWLINE** function in a session applies only to quoted-string literals in SQL statements. The newline mode does not apply to quoted strings contained in host variables in SQL statements. Host variables can contain newline characters within string data regardless of the newline mode currently in effect.

For example, you can use a host variable to insert data that contains newline characters into a column even if the ALLOW_NEWLINE parameter in the `ONCONFIG` file is set to 0.

For further information on how the **IFX_ALLOW_NEWLINE** function affects quoted strings, see Quoted String on page 1202. For further information on the ALLOW_NEWLINE parameter in the `ONCONFIG` file, see the *HCL® Informix® Administrator's Reference*.

User-Defined Functions

A user-defined function (UDF) is a routine that you write in SPL or in a language external to the database, such as C or Java™, and that returns a value to its calling context.

A UDF as an expression has the following syntax:

(explicit id unique_1026_Connect_42_udfu003) unique_1026_Connect_42_udfu003(explicit id unique_1026_Connect_42_udfu004)

unique_1026_Connect_42_udfu004

User-Defined Functions

" *function*("

" [[*parameter* =] <Expression>[]] "

" [, <Statement-Local Variable Declaration>[]] "

") "

Element	Description	Restrictions	Syntax
function	Name of the function	Function must exist	Database Object Name on page 1225
parameter	Name of an argument that was declared in a CREATE FUNCTION statement	If you use the *parameter* = option for any argument in the called function, you must use it for all arguments	Identifier on page 1231

You can call user-defined functions within SQL statements. Unlike built-in functions, user-defined functions can be invoked only by the creator of the function, and by the DBA, and by users who have been granted the Execute privilege on the function. For more information, see Routine-Level Privileges on page 546.

The following examples show some user-defined function expressions. The first example omits the *parameter* option when it lists the function argument:

```
read_address('Miller')
```

This second example uses the *parameter* option to specify the argument value:

```
read_address(lastname = 'Miller')
```

When you use the *parameter* option, the *parameter* name must match the name of the corresponding parameter in the function registration. For example, the preceding example assumes that the **read_address( )** function was registered as follows:

```
CREATE FUNCTION read_address(lastname CHAR(20))
   RETURNING address_t ...  ;
```

A *statement-local variable* (SLV) enables an application to transmit a value from a user-defined function call to another part of the same SQL statement.

To use an SLV with a call to a user-defined function

1. Write one or more OUT parameters (and for UDRs written in the Java™ or in the SPL language, INOUT parameters) for the user-defined function.

   For information about how to write a UDR with OUT or INOUT parameters, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

2. When you register the user-defined function, specify the `OUT` keyword before each OUT parameter, and the `INOUT` keyword before each INOUT parameter.

   For more information, see Specifying INOUT Parameters for a User-Defined Routine on page 1284, and Specifying OUT Parameters for User-Defined Routines on page 1283.

3. Declare the SLV in a function expression that calls the user-defined function with each OUT and INOUT parameter.

   The call to the user-defined function must be made within a WHERE clause. For information about the syntax to declare the SLV, see Statement-Local Variable Declaration on page 1151.

4. Use the SLV that the user-defined function has initialized within the SQL statement.

   After the call to the user-defined function has initialized the SLV, you can use this value in other parts of the same SQL statement in which the SLV was declared, including subqueries of the query whose WHERE clause includes the SLV declaration. For information about the use of an SLV within the SELECT statement, see Statement-Local Variable Expressions on page 1153.

Besides using a SLV to retrieve a value from an OUT or INOUT parameter, you can also use a local variable or a parameter of an SPL routine to retrieve values from an SPL or C routine that has OUT or INOUT parameters.

Statement-Local Variable Declaration

The Statement-Local Variable Declaration declares a statement-local variable (SLV) in a call to a user-defined function that defines one or more OUT or INOUT parameters.

A Statement-Local Variable Declaration has this syntax:

(explicit id unique_1576_Connect_42_slvd001) unique_1576_Connect_42_slvd001(explicit id unique_1576_Connect_42_slvd002)

unique_1576_Connect_42_slvd002

Statement-Local Variable Declaration

" *slv_name* # "

" { <Built-In Data Type>[] | *opaque_data_type* | *distinct_data_type* | <Complex Data Type>[] } "

Element	Description	Restrictions	Syntax
distinct_data_type	Name of a distinct data type	The distinct data type must already exist in the database	Identifier on page 1231

Element	Description	Restrictions	Syntax
opaque_data_type	Name of an opaque data type	The opaque data type must already exist in the database	Identifier on page 1231
slv_name	Name of a statement local variable you are defining	The *slv_name* is valid only for the life of the statement, and must be unique within the statement	Identifier on page 1231

You can declare an SLV in a call to a user-defined function if both of the following conditions are true:

- The UDF has one or more OUT or INOUT parameters
- The SLV is declared when the UDF is invoked in the WHERE clause of a query.

The SLV declaration in the WHERE clause assigns the value of an OUT or INOUT parameter to the SLV, with the sharp (#) symbol between the identifier of the SLV and its declared data type. The UDF can be written in the SPL, C, or Java™ language. For example, if you register a function with the following CREATE FUNCTION statement, you can assign the value of its **y** parameter, which is an OUT parameter, to an SLV in a WHERE clause:

```
CREATE FUNCTION find_location(a FLOAT, b FLOAT, OUT y INTEGER)
   RETURNING VARCHAR(20)
   EXTERNAL NAME "/usr/lib/local/find.so"
LANGUAGE C;
```

In this example, **find_location( )** accepts two FLOAT values that represent a latitude and a longitude and return the name of the nearest city with an extra value of type INTEGER that represents the population rank of the city.

You can now call **find_location( )** in a WHERE clause:

```
SELECT zip_code_t FROM address
   WHERE address.city = find_location(32.1, 35.7, rank # INT)
   AND rank < 101;
```

The function expression passes two FLOAT values to **find_location( )** and declares an SLV named **rank** of type INT. In this case, **find_location( )** will return the name of the city nearest latitude 32.1 and longitude 35.7 (which might be a heavily populated area) whose population rank is between 1 and 100. The statement then returns the zip code that corresponds to that city.

The SLV can be declared only in a call to the UDF in the WHERE clause of the SELECT statement. The scope of reference of the SLV includes other parts of the same SELECT statement. The following SELECT statement, however, is *invalid* because the SLV declaration is in the projection list of the Projection clause, rather than in the WHERE clause:

```
-- invalid SELECT statement
SELECT title, contains(body, 'dog and cat', rank # INT), rank
   FROM documents;
```

The data type that you specify when you declare the SLV must be the same data type as the corresponding OUT or INOUT parameter in the CREATE FUNCTION statement. If you use different but compatible data types, such as INTEGER and FLOAT, the database server automatically performs the cast between the data types.

SLVs share the name space with UDR variables and the column names of the table involved in the SQL statement. Therefore, the database uses the following descending order of precedence to resolve name conflicts among the following objects:

- UDR variables
- Column names
- SLVs

After the call to the UDF assigns the value of an OUT or INOUT parameter to the SLV, you can reference the SLV in other parts of the same query. For more information, see Statement-Local Variable Expressions on page 1153.

Statement-Local Variable Expressions

The Statement-Local Variable Expression specifies a statement-local variable (SLV) that you can use elsewhere in the same SELECT statement.

Statement-Local Variable Expressions

> " *SLV_variable* "

Element	Description	Restrictions	Syntax
SLV_variable	Statement-local variable (SLV) assigned in a call to a user-defined function in the same query	The *SLV_variable* exists only while the query is executing. Its name must be unique within the query	Identifier on page 1231

You define an SLV in the call to a user-defined function in the WHERE clause of the SELECT statement. This user-defined function must be defined with one or more OUT or INOUT parameters. The call to the user-defined function assigns the value of the OUT or INOUT parameters to the SLVs. For more information, see Statement-Local Variable Declaration on page 1151.

Once the user-defined function assigns its OUT or INOUT parameters to the SLVs, you can use these values in other parts of the same SELECT statement, subject to the following scope-of-reference rules:

- The SLV is *read-only* throughout the query (or subquery) in which it is defined.
- The scope of an SLV extends from the query in which the SLV is defined down into all nested subqueries.
- In nested queries, the scope of an SLV does not extend upwards.

  In other words, if a query contains one or more subqueries, an SLV that is defined in the query is also visible to all the subqueries of that query. But if the SLV is defined in the subquery, it is not visible to the parent query.
- In queries that include the UNION operator, the SLV is only visible in the query in which it is defined.

  The SLV is not visible to any other queries specified in the UNION.
- For INSERT, DELETE, and UPDATE statements, an SLV is not visible outside the SELECT portion of the statement.

  Within this SELECT portion of a DML statement, all the above scoping rules apply.

Important: A statement-local variable is in scope only for the duration of a single SQL statement.

The following SELECT statement calls the **find_location( )** function in a WHERE clause and defines the **rank** SLV. Here **find_location( )** accepts two values that represent a latitude and a longitude and return the name of the nearest city with an extra value of type INTEGER that represents the population rank of the city.

```
SELECT zip_code_t FROM address
   WHERE address.city = find_location(32.1, 35.7, rank # INT)
   AND rank < 101;
```

When execution of the **find_location()** function completes successfully, the function has initialized the **rank** SLV. The SELECT then uses this **rank** value in a second WHERE clause condition. In this example, the Statement-Local Variable Expression is the variable **rank** in the second WHERE clause condition:

```
rank < 101
```

The number of OUT and INOUT parameters and SLVs that a UDF can have is not restricted. (Releases of Informix® earlier than Version 9.4 restricted user-defined functions to a single OUT parameter and no INOUT parameters, thereby restricting the number of SLVs to no more than one.)

If the user-defined function that initializes the SLVs is *not* executed in an iteration of the statement, the SLVs each have a value of NULL. Values of SLVs do not persist across iterations of the statement. At the start of each iteration, the database server sets the SLV values to NULL.

The following partial statement calls two user-defined functions with OUT parameters, whose values are referenced with the SLV names **out1** and **out2**:

```
SELECT...
   WHERE func_2(x, out1 # INTEGER) < 100
   AND (out1 = 12 OR out1 = 13)
   AND func_3(a, out2 # FLOAT) = "SAN FRANCISCO"
   AND out2 = 3.1416;
```

If a function assigns one or more OUT or INOUT parameter values from another database of the local database server to SLVs, the values must be of built-in data types, or DISTINCT data types whose base types are built-in data types (and that you cast explicitly to built-in data types), or must be opaque UDTs that you cast explicitly to built-in data types. All the opaque UDTs, DISTINCT types, type hierarchies, and casts must be defined exactly the same way in all of the participating databases.

For more information on how to write a user-defined function with OUT or INOUT parameters, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

Aggregate Expressions

An aggregate expression uses an aggregate function to summarize selected database data. The built-in aggregate functions have the following syntax.

(explicit id unique_1436_Connect_42_aggex003) unique_1436_Connect_42_aggex003(explicit id unique_1436_Connect_42_aggex004)

unique_1436_Connect_42_aggex004

Aggregate Expressions

```
{{COUNT( * ) |{AVG | COUNT | MAX | MIN | SUM | RANGE | STDEV | VARIANCE}({[ <Aggregate Scope
Qualifiers>)]|ALL <Subset of Expression>[] )}}| <User-Defined Aggregates>[]}
```

Aggregate Scope Qualifiers

```
[{ALL | DISTINCT | UNIQUE}][{table . | alias . | view . | synonym .}]column
```

Element	Description	Restrictions	Syntax
column	Column to which aggregate function is applied	See headings for individual keywords on pages that follow	Identifier on page 1231
alias, synonym, table, view	Synonym, table, view, or alias that contains *column*	*Synonym* and the *table* or *view* to which it points must exist	Identifier on page 1231

You cannot use an aggregate expression in a condition that is part of a WHERE clause unless you use the aggregate expression within a subquery. You cannot apply an aggregate function to a BYTE or TEXT column. For other general restrictions, see Subset of Expressions Valid in an Aggregate Expression on page 1155.

An aggregate function returns one value for a set of queried rows. The following examples show aggregate functions in SELECT statements:

```
SELECT SUM(total_price) FROM items WHERE order_num = 1013;

SELECT COUNT(*) FROM orders WHERE order_num = 1001;

SELECT MAX(LENGTH(fname) + LENGTH(lname)) FROM customer;
```

If you use an aggregate function and one or more columns in the projection list of the Projection clause, you must include all the column names that are not used as part of an aggregate or time expression in the GROUP BY clause.

Subset of Expressions Valid in an Aggregate Expression

As indicated in the diagrams for Aggregate Expressions on page 1154 and User-Defined Aggregates on page 1166, not all expressions are available when you use an aggregate expression. The argument of an aggregate function, for example, cannot itself contain an aggregate function. You cannot use aggregate functions in the following contexts:

- In a WHERE clause, but with these two exceptions:
  - unless the aggregate is specified in the Projection clause of a subquery within the WHERE clause,
  - or unless the aggregate is on a correlated column from a parent query, and the WHERE clause is in a subquery within a HAVING clause.
- As an argument to an aggregate function.

  The following nested aggregate expression is not valid:

  ```
  MAX (AVG (order_num))
  ```
- On a column of any of the following data types:
  - Large object (BLOB, BYTE, CLOB, TEXT)
  - Collection data types (LIST, MULTISET, SET)
  - ROW data types (named or unnamed)
  - OPAQUE data types (except with user-defined aggregate functions that support opaque types).

You cannot use a column that is a collection data type as an argument to the following aggregate functions:

- **AVG**
- **SUM**
- **MIN**
- **MAX**

Expression or *column* arguments to built-in aggregates (except for **COUNT**, **MAX**, **MIN**, and **RANGE**) must return numeric or INTERVAL data types, but **RANGE** also accepts DATE and DATETIME arguments.

For **SUM** and **AVG**, you cannot use the difference between two DATE values directly as the argument to an aggregate, but you can use DATE differences as operands within arithmetic expression arguments. For example:

```
SELECT . . .  AVG(ship_date - order_date);
```

returns error -1201, but the following equivalent expression is valid:

```
SELECT . . .  AVG((ship_date - order_date)*1);
```

The following query fragment uses valid syntax to declare aliases for two column expressions:

```
SELECT . . .
   SUM(orders.ship_charge) as o2,
   COUNT(DISTINCT
         CASE WHEN orders.backlog MATCHES 'n'
              THEN orders.order_num END ) AS o3,
   . . .
```

Here the argument to **SUM** is a MONEY(6) column value, and the **COUNT DISTINCT** aggregate takes a **CASE** expression as its argument.

Including or excluding duplicates in the result set

You can use the ALL, DISTINCT, or UNIQUE keywords to qualify the scope of an aggregate function.

If you include an aggregate scope qualifier, it must be the first item in the argument list.

The ALL keyword specifies that all values selected from the column or expression, including any duplicate values, are used in the calculation. Because ALL is the default scope for aggregate functions, the following two aggregate expressions are equivalent, and their return value is based on the value of all qualifying rows in the **ship_weight** column:

```
AVG(ship_weight)
AVG(ALL ship_weight)
```

Including the DISTINCT keyword as the first argument to the aggregate function restricts its subsequent arguments to unique values from the specified column. The UNIQUE and DISTINCT keywords are synonyms in this context. The following two aggregate expressions are equivalent, and their return value is based on the set of unique values in qualifying rows of the **ship_weight** column:

```
AVG(DISTINCT ship_weight)
AVG(UNIQUE ship_weight)
```

If several qualifying rows have the same **ship_weight** value, only one instance of that value is included in calculating the value of the aggregate.

If a query includes the DISTINCT or UNIQUE keyword (rather than the ALL keyword or no keyword) in the Projection clause whose Select list also includes an aggregate function whose argument list begins with the DISTINCT or UNIQUE keyword, the database server issues an error, as in the following example: .

```
SELECT DISTINCT AVG(DISTINCT ship_weight)
       FROM orders;
```

That is, it is not valid in the same query for both the Projection clause and for an aggregate function to restrict the result set to unique values.

If the Projection clause does not specify the DISTINCT or UNIQUE keyword of the SELECT statement, however, the query can include one or more aggregate functions that each includes the DISTINCT or UNIQUE keyword as the first specification in the argument list, as in the following example:

```
SELECT AVG(UNIQUE ship_weight), COUNT (DISTINCT customer_num)
       FROM orders;
```

AVG Function

The **AVG** function returns the average of all values in the specified column or expression.

You can apply the **AVG** function only to number columns. The query in the following example finds the average price of a helmet:

```
SELECT AVG(unit_price) FROM stock WHERE stock_num = 110;
```

The return value is calculated by dividing the sum of **unit_price** values by the cardinality of the qualifying rows.

If you use the DISTINCT or UNIQUE keyword as the first argument, the average (meaning the *mean*) is calculated from only the distinct values in the specified column or expression. In the following example, only one instance of any duplicate values is included when the sum and the cardinality are calculated:

```
SELECT AVG(DISTINCT unit_price) FROM stock WHERE stock_num = 110;
```

If the data set included no duplicate values, both examples above return the same **AVG** value.

NULL values are ignored unless every value in the column or expression is NULL. If every value is NULL, the **AVG** function returns NULL for that column or expression.

Overview of COUNT Functions

The **COUNT** function is actually a set of functions that enable you to count column values in different ways, according to arguments after the COUNT keyword.

Each form of the **COUNT** function is explained in the following subsections. For a comparison of the different forms of the **COUNT** function, see Arguments to the COUNT Functions on page 1159.

COUNT(*) function

The **COUNT (*)** function returns the number of rows that satisfy the WHERE clause of a SELECT statement.

The following example finds how many rows in the **stock** table have the value `HRO` in the **manu_code** column:

```
SELECT COUNT(*) FROM stock WHERE manu_code = 'HRO';
```

The following example queries one of the System Monitoring Interface (SMI) tables to find the number of extents in the **customer** table:

```
SELECT COUNT(*) FROM sysextents WHERE dbs_name = 'stores' AND tabname = customer";
```

You can use **COUNT(*)** as the Projection clause in queries of this general format to obtain information from the SMI tables. For information about **sysextents** and other SMI tables, see the *HCL® Informix® Administrator's Reference* chapter that describes the **sysmaster** database.

If the SELECT statement does not have a WHERE clause, the **COUNT (*)** function returns the total number of rows in the table. The following example finds how many rows are in the **stock** table:

```
SELECT COUNT(*) FROM stock;
```

If the SELECT statement contains a GROUP BY clause, the **COUNT (*)** function reflects the number of values in each group. The following example is grouped by the first name; the rows are selected if the database server finds more than one occurrence of the same name:

```
SELECT fname, COUNT(*) FROM customer GROUP BY fname
   HAVING COUNT(*) > 1;
```

If the value of one or more rows is NULL, the **COUNT (*)** function includes the NULL columns in the count unless the WHERE clause explicitly omits them.

COUNT DISTINCT and COUNT UNIQUE functions

The **COUNT DISTINCT** and **COUNT UNIQUE** functions return unique values.

The **COUNT DISTINCT** function returns the number of unique values in the column or expression, as the following example shows.

```
SELECT COUNT (DISTINCT item_num) FROM items;
```

If the **COUNT DISTINCT** function encounters NULL values, it ignores them unless every value in the specified column is NULL. If every column value is NULL, the **COUNT DISTINCT** function returns zero (0).

The UNIQUE keyword has the same meaning as the DISTINCT keyword in **COUNT** functions. The UNIQUE keyword instructs the database server to return the number of unique non-NULL values in the column or expression. The following example calls the **COUNT UNIQUE** function, but it is equivalent to the preceding example that calls the **COUNT DISTINCT** function:

```
SELECT COUNT (UNIQUE item_num) FROM items;
```

If the Projection clause does not specify the DISTINCT or UNIQUE keyword of the SELECT statement, the query can include multiple COUNT functions that each includes the DISTINCT or UNIQUE keyword as the first specification in the argument list, as in the following example:

```
SELECT COUNT (UNIQUE item_num), COUNT (DISTINCT order_num) FROM items;
```

COUNT column Function

The **COUNT** *column* function returns the total number of non-NULL values in the column or expression, as the following example shows:

```
SELECT COUNT (item_num) FROM items;
```

The ALL keyword can precede the specified column name for clarity, but the query result is the same whether you include the ALL keyword or omit it.

The following example shows how to include the ALL keyword in the **COUNT** *column* function:

```
SELECT COUNT (ALL item_num) FROM items;
```

Arguments to the COUNT Functions

The **COUNT** function accepts as its argument the same expressions that are allowed in the argument list of other built-in aggregate functions, as well as the asterisk (*) notation that only **COUNT** supports. The following categories of built-in expressions are supported as the argument to **COUNT**, as illustrated in the following examples:

- Arithmetic Expressions

  ```
  COUNT(times(informix.sysfragments.evalpos,2))

  SELECT COUNT(a+1), COUNT(2*a), COUNT(5/a), COUNT(times(a, 2)) FROM myTable;
  ```

- Bitwise Logical Functions

  ```
  COUNT(BITAND(informix.systables.flags,1))

  SELECT COUNT(BITAND(a,1)), COUNT(BITOR(8, 20)), COUNT(BITXOR(41, 33)),
         COUNT(BITANDNOT(20,-20)), COUNT(BITNOT(8)) FROM myTable;
  ```

- Cast Expressions

  ```
  COUNT(NULL::int)
  ```

- Conditional Expressions

```
COUNT(CASE WHEN stock.description = "baseball gloves" THEN 1 ELSE NULL END)

SELECT COUNT(CASE WHEN s=14 THEN 1 ELSE NULL END) AS cnt14 FROM all_types;
SELECT COUNT(NVL (ch, 'Addr unk')) FROM all_types;
SELECT COUNT(NULLIF(ch, NULL)) FROM all_types;
```

- Constant Expressions

```
COUNT(CURRENT_ROLE)
COUNT(DATETIME (2007-12-6) YEAR TO DAY)

SELECT COUNT("XX"), COUNT(99),COUNT("t") FROM sysmaster:sysdual;
SELECT COUNT(SET{6, 9, 9, 4}) FROM sysmaster:sysdual;
SELECT COUNT("ROW(7, 3, 6.0, 2.0)") FROM sysmaster:sysdual;
SELECT COUNT(USER), COUNT(CURRENT), COUNT(SYSDATE) from sysmaster:sysdual;
SELECT COUNT(CURRENT_ROLE), COUNT(DEFAULT_ROLE) from sysmaster:sysdual;
SELECT COUNT(DBSERVERNAME), COUNT(TODAY), COUNT(CURRENT) from sysmaster:sysdual;
SELECT COUNT(DATETIME (2007-12-6) YEAR TO DAY) from sysmaster:sysdual;
SELECT COUNT(INTERVAL (16) DAY TO DAY) FROM sysmaster:sysdual;
SELECT COUNT(5 UNITS DAY) FROM sysmaster:sysdual;
```

- Function Expressions

```
COUNT(LENGTH ('abc') + LENGTH (stock.description}
COUNT(DBINFO('sessionid'))
COUNT(user_proc())  --> Here proc() is a user-defined routine.
```

- Column Expressions

```
COUNT(informix.sysfragauth.fragment)
```

You can also use the asterisk (`*`) character, or a column name, or a column name with the ALL, DISTINCT, or UNIQUE aggregate scope qualifiers as the argument to the **COUNT** function to retrieve different types of information about a table. The table below summarizes the meaning of each of the following forms of the **COUNT** function with an asterisk or column name argument.

COUNT Function	**Description**
COUNT (*)	Returns the number of rows that satisfy the query. If you do not specify a WHERE clause, this function returns the total number of rows in the table.
COUNT (DISTINCT) or COUNT (UNIQUE)	Returns the number of unique non-NULL values in the specified column
COUNT (*column*) or COUNT (ALL *column*)	Returns the total number of non-NULL values in the specified column

Some examples can help to show the differences among the various forms of the **COUNT** function that reference a column. Most of the following examples query against the **ship_instruct** column of the **orders** table in the **stores_demo** demonstration database. For information on the schema of the **orders** table and the data values in the **ship_instruct** column, see the description of the demonstration database in the *HCL® Informix® Guide to SQL: Reference*.

Examples of the COUNT(*) Function

In the following example, the user wants to know the total number of rows in the **orders** table. So the user calls the **COUNT(*)** function in a SELECT statement without a WHERE clause:

```
SELECT COUNT(*) AS total_rows FROM orders;
```

The following table shows the result of this query.

total_rows
23

In the following example, the user wants to know how many rows in the **orders** table have a NULL value in the **ship_instruct** column. The user calls the **COUNT(*)** function in a SELECT statement with a WHERE clause, and specifies the IS NULL condition in the WHERE clause:

```
SELECT COUNT (*) AS no_ship_instruct FROM orders
   WHERE ship_instruct IS NULL;
```

The following table shows the result of this query.

no_ship_instruct
2

In the following example, the user wants to know how many rows in the **orders** table have the value `express` in the **ship_instruct** column. So the user calls the **COUNT(*)** function in the projection list and specifies the equals (=) relational operator in the WHERE clause.

```
SELECT COUNT (*) AS ship_express FROM ORDERS
   WHERE ship_instruct = 'express';
```

The following table shows the result of this query.

ship_express
6

Examples of the COUNT DISTINCT Function

In the next example, the user wants to know how many unique non-NULL values are in the **ship_instruct** column of the **orders** table. The user calls the **COUNT DISTINCT** function in the projection list of the SELECT statement:

```
SELECT COUNT(DISTINCT ship_instruct) AS unique_notnulls
   FROM orders;
```

The following table shows the result of this query.

unique_notnulls
16

Examples of the COUNT column Function

In the following example the user wants to know how many non-NULL values are in the **ship_instruct** column of the **orders** table. The user invokes the **COUNT(***column***)** function in the Projection list of the SELECT statement:

```
SELECT COUNT(ship_instruct) AS total_notnulls FROM orders;
```

The following table shows the result of this query.

total_notnulls
21

A similar query for non-NULL values in the **ship_instruct** column can include the ALL keyword in the parentheses that follow the COUNT keyword:

```
SELECT COUNT(ALL ship_instruct) AS all_notnulls FROM orders;
```

The following table shows that the query result is the same whether you include or omit the ALL keyword (because ALL is the default).

all_notnulls
21

MAX Function

The **MAX** function returns the largest value in the specified column or expression.

Using the DISTINCT keyword does not change the results. The query in the following example finds the most expensive item that is in stock but has not been ordered:

```
SELECT MAX(unit_price) FROM stock
   WHERE NOT EXISTS (SELECT * FROM items
      WHERE stock.stock_num = items.stock_num AND
      stock.manu_code = items.manu_code);
```

NULLs are ignored unless every value in the column is NULL. If every column value is NULL, the **MAX** function returns a NULL for that column.

MIN Function

The **MIN** function returns the lowest value in the column or expression. Using the DISTINCT keyword does not change the results. The following example finds the least expensive item in the **stock** table:

```
SELECT MIN(unit_price) FROM stock;
```

NULL values are ignored unless every value in the column is NULL. If every column value is NULL, the **MIN** function returns a NULL for that column.

SUM Function

The **SUM** function returns the sum of all the values in the specified column or expression, as the following example shows:

```
SELECT SUM(total_price) FROM items WHERE order_num = 1013;
```

If you include the DISTINCT or UNIQUE keyword, the returned sum is for only distinct values in the column or expression:

```
SELECT SUM(DISTINCT total_price) FROM items WHERE order_num = 1013;
```

NULL values are ignored unless every value in the column is NULL. If every column value is NULL, the **SUM** function returns a NULL for that column. You cannot use the **SUM** function with a non-numeric column.

RANGE Function

The **RANGE** function returns the range of values for a numeric column expression argument.

It calculates the difference between the maximum and the minimum values, as follows:

```
range(expr) = max(expr) - min(expr);
```

You can apply the **RANGE** function only to numeric columns. The following query finds the range of ages for a population:

```
SELECT RANGE(age) FROM u_pop;
```

As with other aggregates, the **RANGE** function applies to the rows of a group when the query includes a GROUP BY clause, as the next example shows:

```
SELECT RANGE(age) FROM u_pop GROUP BY birth;
```

Because DATE values are stored internally as integers, you can use the **RANGE** function on DATE columns. With a DATE column, the return value is the number of days between the earliest and latest dates in the column.

NULL values are ignored unless every value in the column is NULL. If every column value is NULL, the **RANGE** function returns a NULL for that column.



Important: All computations for the **RANGE** function are performed in 32-digit precision, which should be sufficient for many sets of input data. The computation, however, loses precision or returns incorrect results when all of the input data values have 16 or more digits of precision.

STDEV Function

The **STDEV** function computes the standard deviation of a data set, which is the square root of the **VARIANCE** function. You can apply the **STDEV** function only to numeric columns. The next query finds the standard deviation:

```
SELECT STDEV(age) FROM u_pop WHERE u_pop.age > 0;
```

As with the other aggregates, the **STDEV** function applies to the rows of a group when the query includes a GROUP BY clause, as this example shows:

```
SELECT STDEV(age) FROM u_pop GROUP BY birth WHERE STDEV(age) > 0;
```

NULL values are ignored unless every value in the specified column is NULL. If every column value is NULL, **STDEV** returns a NULL for that column.

Important: All computations for the **STDEV** function are performed in 32-digit precision, which should be sufficient for many sets of input data. The computation, however, loses precision or returns incorrect results when all of the input data values have 16 or more digits of precision.

You cannot use this function on columns of type DATE.

Within a SELECT Statement with GROUP BY clause, **STDEV** returns a zero variance for a count of 1. You can omit this special case through appropriate query construction (for example, "`HAVING COUNT(*) > 1`"). Otherwise, a data set that has only a few cases might block the rest of the query result.

VARIANCE Function

The **VARIANCE** function returns an estimate of the population variance, as the standard deviation squared.

VARIANCE calculates the following value:

$$(SUM(X_i^2) - (SUM(X_i)^2)/N)/(N - 1)$$

In this formula,

- X_i is each value in the column,
- and N is the total number of non-NULL values in the column (unless all values are NULL, in which case the variance is logically undefined, and the **VARIANCE** function returns NULL).

You can apply the **VARIANCE** function only to numeric columns.

The following query estimates the variance of **age** values for a population:

```
SELECT VARIANCE(age) FROM u_pop WHERE u_pop.age > 0;
```

As with the other aggregates, the **VARIANCE** function applies to the rows of a group when the query includes a GROUP BY clause, as in this example:

```
SELECT VARIANCE(age) FROM u_pop GROUP BY birth
   WHERE VARIANCE(age) > 0;
```

As previously noted, **VARIANCE** ignores NULL values unless every qualified row is NULL for a specified column. If every value is NULL, then **VARIANCE** returns a NULL result for that column. (This typically indicates missing data, and is not necessarily a good estimate of underlying population variance.)

If `N`, the total number of qualifying non-NULL column values, equals `1`, then the **VARIANCE** function returns zero (another implausible estimate of the true population variance). To omit this special case, you can modify the query. For example, you might include a `HAVING COUNT(*) > 1` clause.

Important: All calculations for the **VARIANCE** function are performed in 32-digit precision, which should be sufficient for many sets of input data. The calculation, however, loses precision or returns incorrect results when all of the input data values have 16 or more digits of precision.

Although DATE values are stored internally as an integer, you cannot use the **VARIANCE** function on columns of data type DATE.

Error Checking in ESQL/C

Aggregate functions always return exactly one row. If no rows are selected, the function returns a NULL. You can use the **COUNT (*)** function to determine whether any rows were selected, and you can use an indicator variable to determine whether any selected rows were empty. Fetching a row with a cursor that is associated with an aggregate function always returns one row; hence, `100` for end of data is never returned into the **sqlcode** variable for a first FETCH attempt.

You can also use the GET DIAGNOSTICS statement for error checking.

Summary of Aggregate Function Behavior

An example can help to summarize the behavior of the aggregate functions. Assume that the **testtable** table has a single INTEGER column that is named **num**. The contents of this table are as follows.

num
2
2
2
3
3
4
(NULL)

You can use aggregate functions to obtain information about the **num** column and the **testtable** table. The following query uses the **AVG** function to obtain the average of all the non-NULL values in the **num** column:

```
SELECT AVG(num) AS average_number FROM testtable;
```

The following table shows the result of this query.

average_number
2.66666666666667

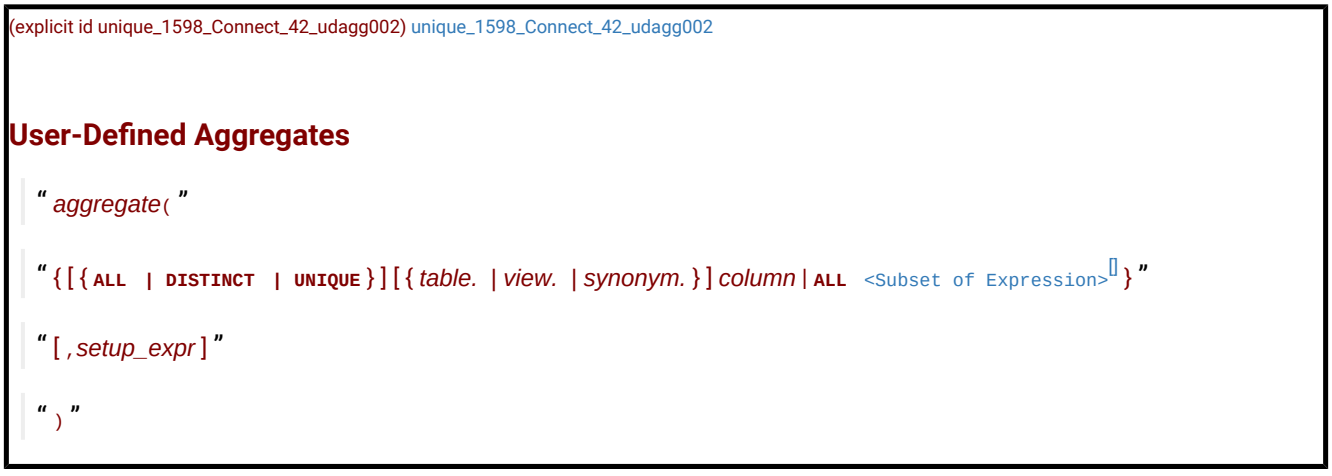
You can use the other aggregate functions in SELECT statements that are similar to the preceding example. If you enter a series of SELECT statements that have different aggregate functions in the projection list and do not include a WHERE clause, you receive the results that the following table shows.

Function	Results	Function	Results
COUNT (*)	`7`	MAX	`4`
COUNT (DISTINCT)	`3`	MAX(DISTINCT)	`4`
COUNT (ALL *num*)	`6`	MIN	`2`
COUNT (*num*)	`6`	MIN(DISTINCT)	`2`
AVG	`2.66666666666667`	RANGE	2
AVG (DISTINCT)	`3.00000000000000`	SUM	`16`
STDEV	0.74535599249993	SUM(DISTINCT)	`9`
VARIANCE	0.55555555555556		

User-Defined Aggregates

You can create your own aggregate expressions with the CREATE AGGREGATE statement and then invoke these aggregates wherever you can invoke the built-in aggregates.

The following diagram shows the syntax for invoking a user-defined aggregate.



Element	Description	Restrictions	Syntax
aggregate	Name of the user-defined aggregate to invoke	The *aggregate* and the support functions defined for *aggregate* must exist	Identifier on page 1231
column	Name of a column within *table*	Must exist and have a numeric data type	Quoted String on page 1202

Element	Description	Restrictions	Syntax
setup_expr	Set-up expression that customizes *aggregate* for a specific invocation	Cannot be a lone host variable. Any columns referenced in *setup_expr* must be in the GROUP BY clause of the query	Expression on page 1003
synonym, *table*, *view*	Synonym, table, or view in which *column* occurs	The *synonym* and the table or view to which it points must exist	Identifier on page 1231

Use the DISTINCT or UNIQUE keywords to specify that the user-defined aggregate is to be applied only to unique values in the named column or expression. Use the ALL keyword to specify that the aggregate is to be applied to all values in the named column or expression.

If you omit the DISTINCT, UNIQUE, and ALL keywords, ALL is the default. For further information on the DISTINCT, UNIQUE, and ALL keywords, see Including or excluding duplicates in the result set on page 1156.

When you specify a setup expression, this value is passed to the **INIT** support function that was defined for the user-defined aggregate in the CREATE AGGREGATE statement.

In the following example, you apply the user-defined aggregate named **my_avg** to all values of the **quantity** column in the **items** table:

```
SELECT my_avg(quantity) FROM items
```

In the following example, you apply the user-defined aggregate named **my_sum** to unique values of the **quantity** column in the **items** table. You also supply the value `5` as a setup expression. This value might specify that the initial value of the sum that **my_avg** will compute is `5`.

```
SELECT my_sum(DISTINCT quantity, 5) FROM items
```

In the following example, you apply the user-defined aggregate named **my_max** to all values of the **quantity** column in the remote **items** table:

```
SELECT my_max(remote.quantity) FROM rdb@rserv:items remote
```

If the **my_max** aggregate is defined as EXECUTEANYWHERE, then the distributed query can be pushed to the remote database server, **rserv**, for execution. If the **my_max** aggregate is not defined as EXECUTEANYWHERE, then the distributed query scans the remote **items** table and computes the **my_max** aggregate on the local database server.

You cannot qualify a user-defined aggregate with the name of a remote database server, as the following example shows. In this case, the database server returns an error:

```
SELECT rdb@rserv:my_max(remote.quantity)
   FROM rdb@rserv:items remote
```

For further information on user-defined aggregates, see CREATE AGGREGATE statement on page 185 and the discussion of user-defined aggregates in *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

Aggregate expressions in grid queries

Aggregate expressions in grid queries require the database server that issues the grid query to combine aggregate results from multiple grid servers.

Grid queries are distributed SELECT statements that return the logical UNION or UNION ALL on qualifying rows of database tables that have identical schemas in databases of a grid. Grid queries are described in the GRID clause on page 728 topic and in the .

Aggregate expressions in grid queries require the database server that issues the Grid query to combine aggregate results from multiple grid servers. Because the Grid query that you specify is converted into a UNION or UNION ALL query across multiple grid servers, each aggregate is calculated independently on each participating server within the specified grid or region. These individual aggregates are returned to the database server that issued the grid query, which calculates their combined UNION or UNION ALL value.

In order to calculate a global aggregation from these individual aggregates, the grid query must be a subquery. For example, suppose we wanted to see for the southeast region (**SW_USA** in the following example) the total sales and the average sale. Because it is not correct to take the average of a group of averages if the number of qualifying rows varies across data sets within the grid, the correct grid query needs to resemble this:

```
SELECT SUM(amt) AS total_sales ,
       SUM(amt) / SUM(cnt) AS avg_sale FROM
 (
 SELECT COUNT(*) cnt, SUM(amt) amt
 FROM sales GRID ALL 'SW_USA'
 );

    total_sales         avg_sale

       $8300.00          $103.75
```

In this example, the projection list of the grid query specifies a COUNT(*) expression (aliased as `cnt`) and an aggregate amount **`SUM(amt)`** (aliased as `amt`) from the **sales** table of each grid server within the southeast region (encoded as `'SW_USA'`). Using these values from each of the remote grid servers, the local grid server can calculate an average for the region. This is accomplished by adding up the total sales by evaluating the expression **`SUM(amt)`** and dividing that value by the total count **`SUM(cnt)`**, where **cnt** and **amt** are columns in the result set of the grid subquery.

Note that the GRID ALL keywords specify a UNION ALL of the qualifying rows from each participating grid server, to avoid eliminating duplicate rows.

OLAP window expressions

You can include On-Line Analytical Processing (OLAP) expressions in a SELECT statement to operate on subsets of the rows in the result set of a query or subquery. You can use OLAP window expressions to examine subsets of the qualifying rows for patterns, trends, or exceptions in the data.

OLAP window expressions allow application developers to compose analytic business queries more easily and efficiently. For example, moving averages and moving sums can be calculated over various intervals. Aggregations and ranks can be reset as selected column values change. Complex ratios can be expressed in simple terms.

Syntax

OLAP window expressions

```
" { <OLAP numbering function> <OVER clause for numbering function> (explicit id ) | <OLAP ranking function>
<OVER clause for ranking functions> (explicit id ) | <OLAP aggregation function> <OVER clause for aggregation
functions> (explicit id ) } "
```

OLAP window expressions are valid in these clauses of the SELECT statement:

- The Select list of the Projection clause
- The ORDER BY clause of the SELECT statement
- Subquery specifications in the Projection clause

OLAP window expressions require an OLAP window function and an OVER clause.

OLAP window function

The OLAP window function identifies an operation for the database server to perform on rows in the query result set. OLAP window functions fit into the following functional categories:

- The OLAP numbering function assigns a unique row number to each row:
  - ROW_NUMBER and ROWNUMBER, which are synonyms
- OLAP ranking functions assign a rank to each row:
  - LAG
  - LEAD
  - RANK
  - DENSE_RANK
  - PERCENT_RANK
  - CUME_DIST
  - NTILE
- OLAP aggregation functions aggregate row data:
  - FIRST_VALUE
  - LAST_VALUE
  - RATIO_TO_REPORT
  - AVG
  - COUNT
  - MAX
  - MIN
  - RANGE
  - STDEV

- SUM
- VARIANCE

OVER clause

The OVER clause defines the result set on which the specified operation is performed. The OVER clause has the following capabilities:

- Define window partitions with the PARTITION BY clause. OLAP window partitions are subsets of the rows in the query result set that are based on the values of one or more column expressions that are listed in the PARTITION BY clause.
- Order rows with the ORDER BY clause. If you include the PARTITION BY clause, you order results within each window partition. Otherwise, you order the whole result.
- Define window frames with the ROWS or RANGE specification for OLAP window aggregation functions. A window frame defines a set of rows within a window partition. The aggregation function operates on the contents of the moving window frame instead of on the whole partition.

For example, the following query contains the OLAP aggregation function SUM:

```
SELECT c,d,
   SUM(d) OVER(
      PARTITION BY a,b
      ORDER BY c,d
      ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
   FROM table1;
```

The PARTITION BY clause creates window partitions for each set of values of columns **a** and **b**.

The ORDER BY clause orders the data within each window partition by the values of the columns **c** and **d**.

The window frame clause that starts with the ROWS keyword creates a window frame that consists of three rows: the current row, the row that precedes the current row, and the row that follows the current row. The current row is the reference point for the window frame. The following table shows how the window frame moves through the result set as each row becomes the current row in turn.

Table 29. Window frames in the result set

Row number	First frame	Second frame	Third frame	Fourth frame	Fifth frame	Sixth frame
1	Current® row	Current® row - 1				
2	Current® row + 1	Current® row	Current® row - 1			
3		Current® row + 1	Current® row	Current® row - 1		
4			Current® row + 1	Current® row	Current® row - 1	
5				Current® row + 1	Current® row	Current® row - 1

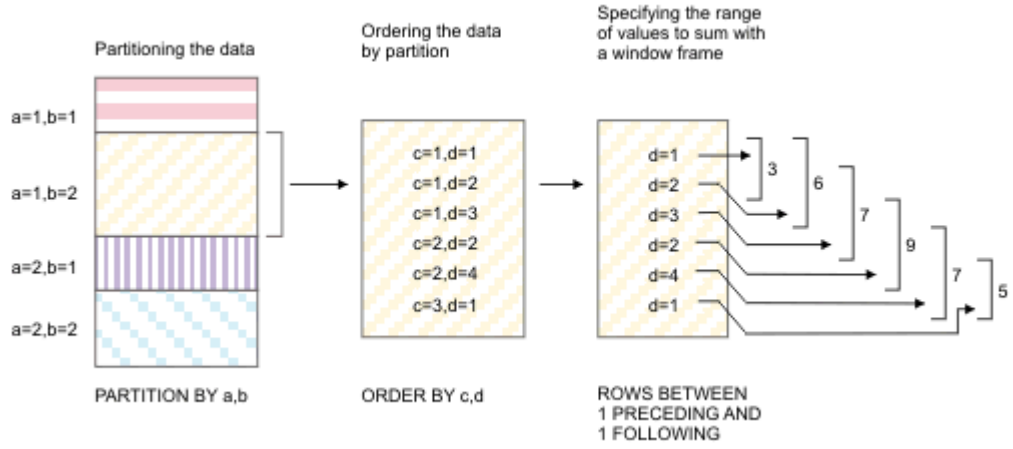
Table 29. Window frames in the result set

(continued)

Row number	First frame	Second frame	Third frame	Fourth frame	Fifth frame	Sixth frame
6					Current® row + 1	Current® row

The SUM function adds the values of **d** column for the three rows for each window partition that is in the scope of the query.

The following illustration shows how the query partitions, orders, and aggregates the data.



The query returns the values of columns **c** and **d** for the single window partition and the sum of up to three values of column **d**:

```
c       d       (sum)
1       1         3
1       2         6
1       3         7
2       2         9
2       4         7
3       1         5
```

The first value in the **sum** column is the sum of the first and second values (1 + 2 = 3) in the **d** column. No preceding value for the current row exists for the first value.

The second value in the **sum** column is the sum of the first, second, and third values (1 + 2 + 3 = 6) in the **d** column.

The third value in the **sum** column is the sum of the second, third, and fourth values (2 + 3 + 2 = 7) in the **d** column.

The last value in the **sum** column is the sum of the fifth and sixth values (1 + 4 = 5) in the **d** column. No following value for the current row exists for the last value.

OLAP numbering function expression

The OLAP numbering function expression returns a sequential number for each row in the result set of a single query.

The OLAP numbering function expression is an OLAP window expression that you can include in the Projection list of a SELECT statement, or the ORDER BY clause of a SELECT statement.

Syntax

OLAP numbering function expression

(explicit id unique_1567_Connect_42_onfl001) unique_1567_Connect_42_onfl001

```
" { ROW_NUMBER | ROWNUMBER } ( ) <OVER clause for the numbering function> "
```

OVER clause for the numbering function

```
" OVER( <Window PARTITION clause>[] <Window ORDER clause>[] ) "
```

Usage

The keywords ROW_NUMBER and ROWNUMBER are synonyms for the same function. This numbering function is like a simplified RANK function that does not require a window ORDER clause and that does not detect duplicate values. The ROW_NUMBER function always returns a unique value for each row in each OLAP window partition.

The ROW_NUMBER function takes no argument, but you must include the empty parentheses after the ROW_NUMBER (or ROWNUMBER) keyword.

The ROW_NUMBER function returns an unsigned integer for every row in each OLAP partition. The sequence of row numbers in each partition starts with 1, and each successive row is incremented by 1, whether consecutive rows in a window partition have the same or different column values.

If the window PARTITION clause is not specified, the complete result set is numbered from 1 to *n*, where *n* is the number of qualifying rows that the query or subquery returns.

The OLAP window that the OVER clause defines for numbering functions has this syntax:

- The window PARTITION clause is optional. If none is specified, the scope of the numbering function is the entire result set of the query or subquery, rather than partitioned subsets.
- The window ORDER clause is optional. If none is specified, the returned row numbers are based on the default order of the rows as processed by the query. If a window ORDER clause is specified, the ORDER BY specification determines the row number assignments.
- The window Frame clause is not supported for OLAP numbering functions.

If the OVER clause that defines the OLAP window for the ROW_NUMBER function omits both the window PARTITION clause and window ORDER clause, you must include the empty parentheses after the OVER keyword.

Example: ROW_NUMBER function

The following query partitions the rows in the product table by distinct package types (pkg_type) and assigns row numbers that restart at 1 for each partition.

```
SELECT ROW_NUMBER()
   OVER(PARTITION BY pkg_type ORDER BY prod_name)
   AS rownum, prod_name, pkg_type
FROM product;

ROWNUM PROD_NAME                     PKG_TYPE
     1 Aroma Sounds CD               Aroma designer box
     2 Aroma Sounds Cassette         Aroma designer box
     1 Christmas Sampler             Gift box
     2 Coffee Sampler                Gift box
     3 Easter Sampler Basket         Gift box
     4 Spice Sampler                 Gift box
     5 Tea Sampler                   Gift box
     1 Aroma Roma                    No pkg
     2 Aroma baseball cap            No pkg
     3 Aroma t-shirt                 No pkg
     4 Assam Gold Blend              No pkg
     5 Assam Grade A                 No pkg
     6 Breakfast Blend               No pkg
     7 Cafe Au Lait                  No pkg
     8 Coffee Mug                    No pkg
     9 Colombiano                    No pkg
    10 Darjeeling Number 1           No pkg
    11 Darjeeling Special            No pkg
    12 Demitasse Ms                  No pkg
    13 Earl Grey                     No pkg
...
```

Because the window ORDER clause specifies the prod_name column as the sorting key, in the default (ASC) order, the returned row numbers are based on the alphabetical order of the product names. For example, the three "Aroma" products are numbered 1, 2, and 3 within the "No pkg" partition. (Uppercase letters sort above lowercase in the default ASCII collation.)

OLAP ranking function expressions

You can include OLAP ranking function expressions to calculate ordinal ranks that can be applied to each row in the partitioned result set of a query or subquery.

OLAP ranking function expressions are OLAP window expressions that you can include in the Projection list of a SELECT statement, or the ORDER BY clause of a SELECT statement.

Syntax

(explicit id unique_1474_Connect_42_orfl001) unique_1474_Connect_42_orfl001(explicit id unique_1474_Connect_42_orfl006)

unique_1474_Connect_42_orfl006(explicit id unique_1474_Connect_42_orfl002) unique_1474_Connect_42_orfl002(explicit id unique_1474_Connect_42_orfl003)

unique_1474_Connect_42_orfl003(explicit id unique_1474_Connect_42_orfl004) unique_1474_Connect_42_orfl004(explicit id unique_1474_Connect_42_orfl005)

unique_1474_Connect_42_orfl005(explicit id unique_1474_Connect_42_orfl007) unique_1474_Connect_42_orfl007

OLAP ranking function expressions

```
" <OLAP ranking function> <OVER clause for ranking functions> "
```

OLAP ranking function

```
" { <LAG function>[] | <LEAD function>[] | <RANK function>[] | <DENSE_RANK function>[] | <PERCENT_RANK function>[]
| <CUME_DIST function>[] | <NTILE function>[] } "
```

OVER clause for ranking functions

```
" OVER( <Window PARTITION clause>[] <Window ORDER clause>[] ) "
```

Usage

The ranking values that these functions return are dependent on the window ORDER clause within the OVER clause. The ORDER clause defines the columns or expressions that the database server uses to calculate the ranking values.

If you omit the window PARTITION clause, the scope of the ranking function is the entire result set of the query or subquery, rather than partitioned subsets of results.

LAG and LEAD functions

The LAG and LEAD functions are OLAP ranking functions that return the value of their *expression* argument for the row at a specified offset from the current row within the current window partition.

Syntax

(explicit id unique_1521_Connect_42_lagle001) unique_1521_Connect_42_lagle001

LAG and LEAD functions

```
" { LAG | LEAD } { ( expression { { RESPECT NULLS | IGNORE NULLS } [ , offset [ , default ] ] ) } | ( expression [ , offset [
, default ] ] ) { RESPECT NULLS | IGNORE NULLS } } "

<OVER clause for Ranking functions>[]
```

Element	Description	Restrictions	Syntax
default	Value to return if *offset* goes beyond the scope of the current window partition	If *default* is not specified, the value of NULL is returned for any out-of-scope row.	Column Expressions on page 1025
expression	Column name, alias, or constant expression to return for the row at *offset* rows from the current row	If *expression* references a column, the column must also be in the select list of the Projection clause.	Column Expressions on page 1025
offset	Non-negative integer constant defining an offset from the position of the current row	Requires a window PARTITION clause. If zero, specifies the current row. If *offset* is not specified, the value `1` is used.	Literal integer

Usage

The *expression* argument is required. The data type of the return value from the LAG or LEAD function is the data type of the expression.

Based on the sort order that the window ORDER clause imposes for each window partition, the LEAD and LAG functions return the value of the expression for every row at *offset* rows from the current row:

- For the LAG function, the *offset* indicates the row that precedes the current row by *offset* rows.
- For the LEAD function, the *offset* indicates the row that follows the current row by *offset* rows.

If the OVER clause includes no window PARTITION clause, these functions return the *expression* values for the entire result set of the query.

If a window PARTITION clause is specified, the second argument to the LAG function (*offset*) means *offset* rows before the current row and within the current partition. For the LEAD function, the second argument means *offset* rows after the current row and within the current partition.

For both functions, if *offset* is not specified, the value `1` is used. If you specify the optional third argument (*default*), which can be an expression, its value is returned if the *offset* goes beyond the scope of the current partition. Otherwise, the NULL value is returned. When the third argument is specified, the second argument must also be specified.

Handling NULL values

In a LAG or LEAD function expression, the optional RESPECT NULLS or IGNORE NULLS keywords can be specified in either of two locations:

- In the argument list
- Immediately following the closing parenthesis that delimits the argument list

The database server issues an exception, however, if you include RESPECT NULLS or IGNORE NULLS in both of these locations within the same LAG or LEAD expression.

The RESPECT NULLS and IGNORE NULLS keywords have these effects:

- If you specify the RESPECT NULLS keywords, rows whose *expression* evaluates to NULL are included when *offset* rows are counted.
- If you specify the IGNORE NULLS keywords, any row whose *expression* evaluates to NULL is not included when *offset* rows are counted.

If you specify IGNORE NULLS and all of the *expression* values for the rows in the window partition are NULL, the LAG or LEAD function returns the *default* value for each row. The function returns the NULL value if no *default* argument is specified.

Example: LEAD and LAG functions

In the following query, the LAG function and the LEAD function each defines an OLAP window that partitions employees by department and lists their salary. The LAG function shows how much more compensation each employee receives, compared to the employee in the same department with the next lower salary value. The LEAD function shows how much less each employee receives, compared to the employee in the same department with the next higher salary value.

```
SELECT name, salary,  LAG(salary)
                 OVER (PARTITION BY dept ORDER BY salary),
                      LEAD(salary, 1, 0)
                 OVER (PARTITION BY dept ORDER BY salary)
FROM employee;

         name      salary     (lag)       (lead)
         John      35,000                 38,400
         Jack      38,400    35,000       41,200
        Julie      41,200    38,400       45,600
        Manny      45,600    41,200       47,300
        Nancy      47,300    45,600       49,500
          Pat      49,500    47,300       51,300
          Ray      51,300    49,500            0
```

The first row with name John has a NULL value for the LAG function because no default value is specified. The last row with name Ray has a 0 value for the LEAD function because 0 is specified as the default value of the third argument in the LEAD function.

RANK function

The RANK function is an OLAP ranking function that calculates a ranking value for each row in an OLAP window. The return value is an ordinal number, which is based on the required ORDER BY expression in the OVER clause.

Syntax

(explicit id unique_1559_Connect_42_knar001) unique_1559_Connect_42_knar001

RANK function

" `RANK ( )` "

" `<OVER clause for ranking functions>` [] "

Usage

The rank of a row is defined as 1 plus the number of rows whose rankings precede that of the row. If two or more rows have the same value, these rows get the same rank as well. Results can have a gap in the sequence of consecutive ranked values. For example, if two rows are ranked 1, the next ranking is 3. The DENSE_RANK function uses a different rule for ranking rows that include non-unique values.

The RANK function takes no argument, but the empty parentheses must be specified. If the OVER clause specifies the optional window PARTITION clause, the rankings are calculated within the subset of rows that each partition defines.

Example

Example: RANK function

The following query ranks sales people by the amount of their sales. The ranks are not consecutive because ties in sales amounts are both assigned the same rank value, and the next rank value is skipped.

```
SELECT emp_num, sales,
  RANK() OVER (ORDER BY sales) AS rank
  FROM sales;

   emp_num      sales   rank
        101      2,000      1
        102      2,400      2
        103      2,400      2
        104      2,500      4
        105      2,500      4
        106      2,650      6
```

DENSE_RANK function

The DENSE_RANK function is an OLAP ranking function that calculates a ranking value for each row in an OLAP window. The return value is an ordinal number, which is based on the required ORDER BY expression in the OVER clause.

Syntax

(explicit id unique_1503_Connect_42_drank001) unique_1503_Connect_42_drank001

DENSE_RANK function

" { DENSE_RANK | DENSERANK } ( ) "

" <OVER clause for Ranking functions> [] "

Usage

The rank of a row is defined as 1 plus the number rankings that precede the ranking of the row. If two or more rows have the same value, these rows get the same rank. However, in contrast to the RANK function, if two or more rows tie, there is no gap in the sequence of ranked values. For example, if two rows are ranked 1, the next ranking is still 2.

This function takes no argument, but the empty parentheses must be specified. If the OVER clause specifies the optional window PARTITION clause, the DENSE_RANK rankings are calculated within the subset of rows that each window partition defines.

Example

Example: DENSE_RANK function

The following query ranks sales people by the amount of their sales. Ranks are consecutive even if multiple sales amounts have the same rank.

```
SELECT emp_num, sales,
  DENSE_RANK() OVER (ORDER BY sales) AS dense_rank,
  FROM sales;

   emp_num      sales   dense_rank

        101      2,000      1
        102      2,400      2
        103      2,400      2
        104      2,500      3
        105      2,500      3
        106      2,650      4
```

PERCENT_RANK function

The PERCENT_RANK function is an OLAP ranking function that calculates a ranking value for each row in an OLAP window, normalized to a range from 0 to 1.

Each PERCENT_RANK value is computed as the row's RANK minus 1, divided by the number of rows in the partition minus 1. Values closer to 1 generally represent higher rankings and values closer to 0 generally represent lower rankings.

Syntax

(explicit id unique_1553_Connect_42_pctr004) unique_1553_Connect_42_pctr004

PERCENT_RANK function

```
PERCENT_RANK ( )  <OVER clause for Ranking functions>[]
```

Usage

This function takes no argument, but the empty parentheses must be specified. If the optional window PARTITION clause is also specified, the rankings are calculated within the subset of rows that each partition defines. If there is a single row in the partition, its PERCENT_RANK value is 0.

Example

Example: PERCENT_RANK function

The following query ranks sales people by the amount of their sales.

```
SELECT emp_num, sales,
  PERCENT_RANK() OVER (ORDER BY sales) AS per_rank
FROM sales;

    emp_num      sales  per_rank

        101      2,000         0
        102      2,400       0.2
        103      2,400       0.2
        104      2,500       0.6
        105      2,500       0.6
        106      2,650       1.0
```

CUME_DIST function

The CUME_DIST function is an OLAP ranking function that calculates a cumulative distribution as a percentile ranking for each row. The rank is expressed as a decimal fraction that ranges from 0 to 1.

Syntax

(explicit id unique_1475_Connect_42_cume004) unique_1475_Connect_42_cume004

CUME_DIST function

```
CUME_DIST ( )  <OVER clause for Ranking functions>[]
```

Usage

The CUME_DIST function calculates the number of rows that are ranked lower than or equal to the current row, including the current row, which is divided by the total number of rows in the partition. Values closer to 1 represent higher rankings and values closer to 0 represent lower rankings.

This function takes no argument, but the empty parentheses must be specified. If the optional window PARTITION clause is also specified, the rankings are calculated within the subset of rows that each partition defines. If there is a single row in the partition, its CUME_DIST value is 1.

Example

Example: CUME_DIST function

The following query shows the cumulative distribution of the amount of sales per sales person.

```
SELECT emp_num, sales,
  CUME_DIST() OVER (ORDER BY sales) AS cume_dist
FROM sales;

    emp_num      sales     cume_dist

        101      2,000    0.166666667
        102      2,400    0.500000000
        103      2,400    0.500000000
        104      2,500    0.833333333
        105      2,500    0.833333333
        106      2,650    1.000000000
```

NTILE function

The NTILE function is an OLAP ranking function that classifies the rows in each partition into N ranked categories, called tiles, where each category includes an approximately equal number of rows.

Syntax

(explicit id unique_1547_Connect_42_ntilel001) unique_1547_Connect_42_ntilel001

NTILE function

" **NTILE** (*unsigned*) "

<OVER clause for Ranking functions>[]

Element	Description	Restrictions	Syntax
unsigned	Unsigned integer that specifies how many categories, or tiles, to rank	Cannot be zero	Literal integer

Usage

The number of ranked categories, or tiles, is set by the unsigned integer argument to the function, and on the ORDER BY expression in the OVER clause.

For example, the following ranking function expression returns percentile rankings from 1 to 100 for the values in the **dollars** column for all the rows in the query result set:

```
NTILE(100) OVER(ORDER BY dollars)
```

When the argument is `4`, the returned values sort the rows in each partition that the OVER clause defines into four quartiles. When a set of values is not divisible by the specified integer argument, the NTILE function puts leftover rows in the lower-ranked tiles.

Example

Example: NTILE function

The following query ranks employees in departments by employee salary, and calculates the tile number of 1 through 5 for each department.

```
SELECT name, salary,
   NTILE(5) OVER (PARTITION BY dept ORDER BY salary)
FROM employee;
         name      salary        (ntile)
         John      35,000              1
         Jack      38,400              1
        Julie      41,200              2
        Manny      45,600              2
        Nancy      47,300              3
          Pat      49,500              4
          Ray      51,300              5
```

The salaries are ordered from lowest to highest because the default ordering direction for the ORDER BY clause is ascending. If you include the DESC keyword in the ORDER BY clause, the salaries are ordered from highest to lowest.

OLAP aggregation function expressions

OLAP aggregation function expressions can return aggregate information about rows in the partitioned results of a query.

OLAP aggregation function expressions are OLAP window expressions that you can include in the Projection list of a SELECT statement, or the ORDER BY clause of a SELECT statement.

Syntax

(explicit id unique_1510_Connect_42_oafl001) unique_1510_Connect_42_oafl001(explicit id unique_1510_Connect_42_orfl002) unique_1510_Connect_42_orfl002(explicit id unique_1510_Connect_42_orfl003) unique_1510_Connect_42_orfl003(explicit id unique_1510_Connect_42_orfl005) unique_1510_Connect_42_orfl005(explicit id unique_1510_Connect_42_oafl004) unique_1510_Connect_42_oafl004

OLAP Aggregation function expressions

" <Aggregation functions> <OVER clause for Aggregation functions> "

OLAP Aggregation functions

" { <FIRST_VALUE function> [] | <LAST_VALUE function> [] | <RATIO_TO_REPORT function> [] | <OLAP Window Aggregate functions> [] } "

OVER clause for Aggregation functions

" **OVER** [] (<Window PARTITION clause> [] "

" [<Window ORDER clause> [] <Window Frame clause> []]) "

Usage

You can apply the OLAP aggregation functions to subsets of the rows in an OLAP window partition by specifying a window frame.

LAST_VALUE function

The LAST_VALUE window aggregation function returns the value of a specified expression for the last row in each OLAP window partition.

Syntax

(explicit id unique_1511_Connect_42_lastval001) unique_1511_Connect_42_lastval001

LAST_VALUE function

" **LAST_VALUE** { (*expression*) { { **RESPECT NULLS** | **IGNORE NULLS** } } | (*expression* { **RESPECT NULLS** | **IGNORE NULLS** }) } "

<OVER clause for aggregation functions> []

Element	Description	Restrictions	Syntax
expression	Column name, alias, or constant expression	If *expression* references a column, the column must also be in the select list of the Projection clause	Column Expressions on page 1025

Usage

The return data type of the LAST_VALUE function is the data type of the specified expression. The result can be NULL. If IGNORE NULLS is specified, all rows where the expression value for the row evaluates to a NULL value are not considered in the calculation. If IGNORE NULLS is specified and all values in the OLAP window are NULL, the LAST_VALUE function returns the NULL value.

The RESPECT NULLS or IGNORE NULLS option can be specified either within the parentheses immediately following the expression, or outside the parentheses, but only one such specification is allowed.

RATIO_TO_REPORT function

The RATIO_TO_REPORT function calculates the fractional ratio of each row to the rest of the rows in the window partition, based on the numeric argument to the function.

Syntax

(explicit id unique_1560_Connect_42_rat2r001) unique_1560_Connect_42_rat2r001

RATIO_TO_REPORT function

" { **RATIO_TO_REPORT | RATIOTOREPORT** } (*expression*) "

" <OVER clause for aggregation functions> [] "

Element	Description	Restrictions	Syntax
expression	Column name, alias, or constant expression	The *expression* must evaluate to a numeric data type. DATE, DATETIME, and INTERVAL columns are not valid. If *expression* references a column, the column must also be in the select list of the Projection clause.	Column Expressions on page 1025

Usage

This function calculates the ratio of values for a specified numeric column in a row versus the sum of values from all rows in each partition in an OLAP window frame. The name RATIOTOREPORT is a keyword synonym for RATIO_TO_REPORT.

As with all OLAP window aggregation functions, the Window PARTITION, Window ORDER, and Window Frame clauses are optional. The ratios can be applied either to partitioned rows or to the complete query result set. If a window Frame clause is also specified, then the calculated ratio applies to all rows from the current window frame.

The required *expression* argument must specify a column of a numeric data type, or else be a constant expression that evaluates to a numeric data type. If *expression* is not a numeric data type, the function fails with Error Message -25862.

If the sum of *expression* values for all the rows in the current window partition is zero, this function returns the NULL value for each row in that partition.

If another OLAP aggregation function, such as SUM or MAX, has an empty OVER clause, or if the OVER clause contains only a single window PARTITION specification, the result of the OLAP function is the same for every row in the partition. This is not the case, however, for RATIO_TO_REPORT, because different rows in the same window partition are assigned different ratios, totaling 1 (approximately) for each partition. To convert ratios to percentages, multiply the function expression by 100, as in the following example.

Example: RATIO_TO_REPORT function

The following example calculates the decimal fraction of each city's sales, based on all rows that the query returns, as a single report that shows the sales totals for each city, in descending order.

```
SELECT city, SUM(dollars) AS SALES,
   RATIO_TO_REPORT(SUM(dollars)) OVER() *100 AS RATIO_DOLLARS
   FROM sales, store, period
   WHERE sales.store_id = store.store_id
      AND sales.period_id = period.period_id
   GROUP BY city
   ORDER BY sales DESC;
```

```
CITY               SALES           RATIO_DOLLARS
San Jose           896931.15          12.58
Atlanta            514830.00           7.22
Miami              507022.35           7.11
Los Angeles        503493.10           7.06
Phoenix            437863.00           6.14
New Orleans        429637.75           6.03
Cupertino          424215.00           5.95
Boston             421205.75           5.91
Houston            417261.00           5.85
New York           397102.50           5.57
Los Gatos          394086.50           5.53
Philadelphia       392377.75           5.50
Milwaukee          389378.25           5.46
Detroit            305859.75           4.29
Chicago            294982.75           4.14
Hartford           236772.75           3.32
Minneapolis        165330.75           2.32
```

In the example above, the *expression* argument to RATIO_TO_REPORT is the numeric aggregate function expression `SUM(dollars)`. The last row of the output indicates that the **sales** value for the **city** Minneapolis is approximately 2.32% of the total sales that the query reports.

OLAP window aggregate functions

Several functions that return aggregate results, such as sums and averages, from the results of a query can also be used as OLAP functions from the context of an OLAP window.

Syntax

(explicit id unique_1601_Connect_42_waf004) unique_1601_Connect_42_waf004

Window aggregate functions

" { **AVG** | **COUNT** | **MAX** | **MIN** | **RANGE** | **STDEV** | **SUM** | **VARIANCE** } ([{ **DISTINCT** | **UNIQUE** }] [[{ *table* | *alias* | *view* | *synonym* }] .] *column_expr*) "

<OVER clause for aggregation functions>[]

Element	Description	Restrictions	Syntax
column_expr	Column expression argument to the aggregate function	See the headings for individual functions below	Identifier on page 1231
alias, *synonym*, *table*, *view*	Synonym, table, view, or alias that contains *column*	*Synonym* and the *table* or *view* to which it points must exist	Identifier on page 1231

Usage

These aggregate functions do not require an OLAP window to calculate aggregate values from a query result set. They behave like OLAP window aggregation functions, however, in calling contexts where the OVER clause in the function expression defines one or more window partitions, or includes the window ORDER clause and the window Frame clause.

Important: When the DISTINCT or UNIQUE keyword is part of the window aggregate function specification, the OVER clause of the window aggregate expression cannot include the Window ORDER clause or the Window Frame clause.

The following aggregate functions can return information about the rows in OLAP window partitions.

AVG function

The AVG function returns the average of all values in the specified column or expression in a window partition of the query results, for the rows in each partition that is defined in the OVER clause. If the OVER clause includes the Window Frame clause, AVG returns a value for each set of rows in a window frame.

You can apply the AVG function only to columns of number data types. If you use the DISTINCT (or UNIQUE) keyword, the average (meaning the mean) is calculated from only the distinct values in the specified column or expression, and the OVER clause cannot include a Window ORDER or Window Frame clause.

NULL values are ignored unless every value in the column or expression is NULL. If every value is NULL, the AVG function returns a NULL for that column or expression.

You cannot use the AVG function with a nonnumeric column or expression.

COUNT function

The COUNT function returns the cardinality of non-NULL values in the specified column or expression in a window partition of the query results, for each partition that is defined in the OVER clause. If the OVER clause includes the Window Frame clause, COUNT returns a value for each set of rows in a window frame.

If you use the DISTINCT (or UNIQUE) keyword, the cardinality of rows in the partition is calculated from only the distinct values in the specified column or expression, and the OVER clause cannot include a Window ORDER or Window Frame clause.

NULL values are ignored unless every value in the column or expression is NULL. If every value is NULL, the COUNT function returns a NULL for that column or expression.

MAX function

The MAX function returns the largest value in the column or expression in a window partition of the query results, for the rows in each partition that is defined in the OLAP window OVER clause.

Specifying the DISTINCT or UNIQUE keyword has no effect on the results, but (as with the other built-in aggregate functions) no Window ORDER or Window Frame clause is allowed.

If the OVER clause includes the Window Frame clause, MAX returns a value for each set of rows in a window frame.

When a column expression is specified as argument to COUNT, NULL values are ignored unless every value in the specified column expression is NULL. If every value is NULL, MAX returns a NULL value for that column or expression. When COUNT(*) is specified, NULL values are counted the same as other values.

MIN function

The MIN function returns the lowest value in the column or expression in a window partition of the query results, for each partition that the OLAP window OVER clause defines. If the OVER clause includes the Window Frame clause, the MIN function returns a value for each set of rows.

Specifying the DISTINCT or UNIQUE keyword has no effect on the results, but (as with the other built-in aggregate functions) no Window ORDER or window Frame clause is allowed.

If the OVER clause includes the Window Frame clause, MIN returns a value for each set of rows in a window frame.

NULL values are ignored unless every value in the specified column expression is NULL. If every value is NULL, MIN returns a NULL value for that column expression.

RANGE function

The RANGE function returns the range of values in the column or expression in a window partition of the query results, for each partition that the OLAP window OVER clause defines. If the OVER clause includes the Window Frame clause, RANGE returns a value for each set of rows in a window frame.

The RANGE function calculates the difference between the maximum and the minimum values, as follows:

```
range(expr) = max(expr) - min(expr)
```

You can apply the RANGE function only to numeric column expressions. The following query finds the range of ages for a population:

```
SELECT RANGE(age) OVER () FROM u_pop;
```

Because DATE values are stored internally as integers, you can use the RANGE function on DATE column. With a DATE column, the return value is the number of days between the earliest and latest dates in the column expression.

NULL values are ignored unless every value in the column expression is NULL. If every column expression value is NULL, the RANGE function returns a NULL for that column expression.

STDEV function

The STDEV function returns the standard deviation of a column or expression, using the following formula:

$$SQRT((SUM(X_i^2) - (SUM(X_i)^2)/N)/(N - 1))$$

In this formula, X_i is each column expression value in the window partition or frame that the OVER clause specifies, and N is the total number of non-NULL values in the column expression.

If the OVER clause includes the Window Frame clause, the STDEV function returns a value for each set of rows in a window frame.

NULL values are ignored, unless every value in the specified column expression is NULL. If every column expression value is NULL, the STDEV function returns NULL for that column expression.

You can apply the STDEV function only to numeric column expressions. You cannot use this function on column expressions of type DATE.

SUM function

The SUM function calculates and returns the sum of all the values of a column expression in a window partition of the query results, for the rows in each partition that is defined in the OLAP window OVER clause.

If you use the DISTINCT (or UNIQUE) keyword as the first item in the argument list, the sum is for only distinct values in the column or expression, and no Window ORDER or Window Frame clause is allowed.

NULL values are ignored, unless every value is NULL. If every value is NULL, the SUM function returns the NULL value for that column or expression.

You cannot use the SUM function with a nonnumeric column or expression.

VARIANCE function

The VARIANCE function calculates and returns the mean squared error as an estimate of the population variance of the values in a specified numeric column or expression for the partitions of query results that are defined in the OVER clause that follows the OLAP window VARIANCE expression.

If the OVER clause includes the Window Frame clause, the VARIANCE function returns a value for each set of rows, using the following formula:

```
(SUM(Xi2) - (SUM(Xi)2)/N)/(N - 1)
```

In this formula,

- X_i is each column value in the window partition or frame that the OVER clause specifies,
- and N is the total number of non-NULL values in the column (unless all values are NULL, in which case the variance is logically undefined, and the VARIANCE function returns NULL).

You can apply the VARIANCE function only to numeric columns.

Example

Example: AVG function with partitioning

In the following example, the AVG function is used in an OLAP window aggregation expression to return the rolling average **closeprice** column values during the year 2012 for two window partitions, based on `ABC` and on `XYZ` values in the **symbol** column as the partitioning keys.

```
SELECT symbol, tradingdate,
  AVG(closeprice) OVER (PARTITION BY symbol
    ORDER BY tradingdate
  ROWS BETWEEN 29 PRECEDING AND CURRENT ROW)
FROM dailystockdata
WHERE symbol IN ('ABC', 'XYZ')
  AND tradingdate BETWEEN '2012-01-01' AND '2012-12-31';
```

The window ORDER clause specifies the **tradingdate** column value as the sorting key, and the window Frame clause defines a rolling window that is based on the average of 30 consecutive **tradingdate** values, ending with the **tradingdate** of the current row.

Example

Example: AVG function without partitioning

The following query returns the stock price ordered by day and the average of the prices of the current day, the day before, and the day after. The result set is not partitioned because the query does not include a PARTITION BY clause.

```
SELECT price,
       AVG(price) OVER (ORDER BY tradingday
                  ROWS BETWEEN 1 PRECEDING AND 1 FOLLOWING)
FROM stock_price
WHERE tradingday BETWEEN '2012-11-01' AND '2012-11-07';
```

```
         price          (avg)
     18.25     18.31
     18.37     18.31
               18.37
               19.03
     19.03     18.81
     18.59     18.61
     18.21     18.40
```

The first value in the **avg** column is the average of the first two values in the **price** column, because no preceding values exist for the first value of the **price** column.

The second value in the **avg** column is the average of the first two values in the **price** column, because the third row of the **price** column does not have a value.

The third value in the **avg** column is equal to the second value in the **price** column because the third and fourth rows in the **price** column do not have values.

Example

Example: COUNT function

The following query returns the shipping date, the shipping charge, and the number of orders for each order by customer. The query results are partitioned by customer number and limited to customer numbers that are less than or equal to 110.

```
SELECT customer_num, ship_date, ship_charge,
       COUNT(*) OVER (PARTITION BY customer_num)
FROM orders
WHERE customer_num <= 110;

customer_num ship_date  ship_charge       (count(*))

         101 05/26/2008      $15.30               1
         104 05/23/2008      $10.80               4
         104 07/03/2008       $5.00               4
         104 06/01/2008      $10.00               4
         104 07/10/2008      $12.20               4
         106 05/30/2008      $19.20               2
         106 07/03/2008      $12.30               2
         110 07/06/2008      $13.80               2
         110 07/16/2008       $6.30               2
```

Customer 104 appears in the list four times. The value in the **count** column is always 4 for customer 104.

OVER clause for OLAP window expressions

The OVER clause defines the result set on which an OLAP window expression is performed.

Syntax

OVER clause

"{ <OVER clause for numbering functions> <OVER clause for ranking functions> <OVER clause for aggregation functions> }"

OVER clause for numbering functions

"**OVER**(<Window PARTITION clause> <Window ORDER clause>)"

OVER clause for ranking functions

"**OVER**(<Window PARTITION clause> <Window ORDER clause>)"

OVER clause for aggregation functions

"**OVER**(<Window PARTITION clause>"

"[<Window ORDER clause> <Window Frame clause>])"

Window PARTITION clause

"**PARTITION BY** *partition_key*"

Window ORDER clause

"**ORDER BY** *sorting_key* [{ **ASC** | [**DESC**] }] [**NULLS** { **FIRST** | **LAST** }]"

Window Frame clause

"{ **RANGE** | **ROWS** } { { **UNBOUNDED PRECEDING** | *offset***PRECEDING** | **CURRENT ROW** } | **BETWEEN** { **UNBOUNDED PRECEDING** | *offset***PRECEDING** | *offset***FOLLOWING** | **CURRENT ROW** } **AND** { **UNBOUNDED FOLLOWING** | *offset***PRECEDING** | *offset***FOLLOWING** | **CURRENT ROW** } }"

Element	Description	Restrictions	Syntax
offset	Unsigned integer that represents the offset from the position of the current row	Cannot be negative. If zero, specifies the current row	Literal integer.
partition_key	Column name, alias, or constant expression by which to partition rows	Must be in the select list of the Projection clause	Column Expressions on page 1025
sorting_key	Column name, alias, or constant expression by which to sort rows	Same restrictions as for *partition_key*. For RANGE window frames, only a single sorting key is allowed, and the data type must be numeric, DATE, or DATETIME.	Column Expressions on page 1025

If the OVER clause is empty, you must still include the empty parentheses.

Window PARTITION clause

An OLAP window partition is a subset of the rows that are returned by a query. Each partition is defined by one or more column expressions in the PARTITION BY specification of the OVER clause that defines the window. The database server applies the specified OLAP window function to all of the rows in each window partition. If no partitions are defined in the OVER clause, the window function is applied to every row in the result set of the query.

Window ORDER clause

The database server sorts the rows in each window partition according to the sort key (or multiple sort keys) in the window ORDER clause. If you specify no ascending (ASC) or descending (DESC) order, ASC is the default. If no ORDER clause is specified, the order of the qualifying rows is the order in which the rows were retrieved.

Window Frame clause

The window Frame clause can return subsets, called *aggregation groups*, of the rows in each window partition. A window frame is defined by a specific number of rows or by a range of values.

Row-based window frame

The ROWS keyword creates a row-based window frame that consists of a specific number of rows that precede or follow the current row, or both. The offset represents the number of rows to return. The following example returns seven rows that include the six rows that precede the current row:

```
AVG(price) OVER (ORDER BY year, day
   ROWS BETWEEN 6 PRECEDING AND CURRENT ROW)
```

In a row-based window frame clause, offsets are expressed as unsigned integers because the keyword FOLLOWING specifies a positive offset from the current rows, and the keyword PRECEDING specifies a negative offset from the current row. The keyword UNBOUNDED refers to all of the rows from the current row to the limit of the window partition. As the first term after the ROWS keyword in a window Frame specification,

UNBOUNDED PRECEDING means that the starting boundary is the first row in the partition, and UNBOUNDED FOLLOWING means that the ending boundary is the last row in the partition.

Value-based window frame

The RANGE keyword creates a value-based frame clause that consists of the current row plus the rows that meet the criteria that is set by the sorting key in the ORDER clause and fit into the specified offset. The offset represents the number of units of the data type of the sorting key. The sorting key must be a numeric, DATE, or DATETIME data type. For example, if the sorting key is a DATE data type, the offset represents a specific number of days. The following example returns the count of the number of rows that have a ship date within 2 days of the current row plus the current row:

```
COUNT(*) OVER (ORDER BY ship_date
   RANGE BETWEEN 2 PRECEDING AND 2 FOLLOWING)
```

Value-based window frames define rows within a window partition that contain a specified range of numeric values. The window ORDER clause of the OVER function defines the numeric, DATE, or DATETIME column to which the RANGE specification is applied, relative to the current row value for that column. Only a single sorting key is allowed in the ORDER clause of value-based window frames.

In both the row-based and value-based cases, the OLAP function is calculated on the contents of this window frame, rather than the fixed contents of the whole partition. The window frame does not need to contain the current row. For example, the following specification defines a window frame that contains only the row before the current row:

```
ROWS BETWEEN 1 PRECEDING AND 1 PRECEDING
```

If you specify no window ORDER clause for a window aggregation function, then by default, the result set is not restricted, which is equivalent to the following window frame specification:

```
ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING
```

If you specify an ORDER clause but no window frame clause for a window aggregation function, then by default, all rows that precede the current row and the current row are returned, which is equivalent to the following window frame specification:

```
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
```

Example

Example: SUM function without a window frame

The following query returns the sales by quarter for one year and the cumulative sum of the sales by quarter.

```
SELECT sales, SUM(sales) OVER (ORDER BY quarter)
   FROM sales WHERE year = 2013
       sales     (sum)
         120       120
         135       255
         127       382
         153       535
```

The sum of the sales for the fourth quarter is equal to the sales in all four quarters.

Because the query does not include a window frame clause, the SUM function operates on the entire result set, as specified by the FROM clause.

Example

Example: Row-based window frame

The following query returns players that are partitioned by teams and ordered by the number of points. Within each partition, the number of points for the player and the previous player are averaged:

```
SELECT team, player, points,
   AVG(points) OVER(PARTITION BY team ORDER BY points
      ROWS 1 PRECEDING AND CURRENT ROW) AS olap_avg
FROM points;

TEAM       PLAYER     POINTS OLAP_AVG
A          Singh           7  7.00000000000
A          Smith          14 10.50000000000
B          Osaka           8  8.00000000000
B          Ricci          12 10.00000000000
B          Baxter         18 15.00000000000
C          Chun           13 13.00000000000
D          Kwan            9  9.00000000000
D          Tran           16 12.50000000000
```

Example

Example: Range-based window frame

The following query returns players that are partitioned by teams and ordered by age. Within each partition, the number of points for each player and any player who is up to 9 years older is averaged:

```
SELECT player, age, team, points,
   AVG(points) OVER(PARTITION BY team ORDER BY age
      RANGE BETWEEN CURRENT ROW AND 9 FOLLOWING) AS olap_avg
FROM points_age;

PLAYER     AGE TEAM       POINTS OLAP_AVG
Singh       25 A               7 10.50000000000
Smith       26 A              14 14.00000000000
Baxter      27 B              18 13.00000000000
Osaka       35 B               8 10.00000000000
Ricci       40 B              12 12.00000000000
Chun        21 C              13 13.00000000000
Kwan        22 D               9 12.50000000000
Tran        31 D              16 16.00000000000
```

In partition A, the average for Singh includes the points for Smith, because Smith is one year older than Singh. The average for Smith does not include the points from Singh, because Singh is younger than Smith.

In partition B, the average for Baxter includes the points for Osaka, who is 8 years older than Baxter, but not for Ricci, who is 13 years older than Baxter.

In partition D, the average for Kwan includes the points for Tran, because Tran is 9 years older than Kwan.

Example: Window frame without the current row

The following query calculates the average number of points for the preceding two rows in the partition:

```
SELECT player, age, team, points,
   AVG(points) OVER(PARTITION BY team ORDER BY age
   ROWS BETWEEN 2 PRECEDING AND 1 PRECEDING) AS olap_avg
   FROM points_age;
PLAYER     AGE  TEAM   POINTS   OLAP_AVG
Singh        25 A             7             NULL
Smith        26 A            14    7.00000000000
Baxter       27 B            18             NULL
Osaka        35 B             8   18.00000000000
Ricci        40 B            12   13.00000000000
Chun         21 C            13             NULL
Kwan         22 D             9             NULL
Tran         31 D            16    9.00000000000
```

In partition B, the average for Ricci is based on the points totals for Baxter and Osaka: (18 + 8 = 26)/2 = 13. When the current row has no preceding rows to use for the calculation, the result is NULL.

INTERVAL Field Qualifier

The INTERVAL field qualifier specifies the precision, in time units, for an INTERVAL value. Use the INTERVAL Field Qualifier segment whenever you see a reference to an INTERVAL field qualifier in a syntax diagram.

Syntax

INTERVAL Field Qualifier

" { { **DAY** *[(precision)]* { **TO DAY** | **TO HOUR** | **TO MINUTE** | **TO SECOND** | **TO FRACTION** [{ *(scale)* }] } | **HOUR** *[(precision)]* { **TO HOUR** | **TO MINUTE** | **TO SECOND** | **TO FRACTION** [{ *(scale)* }] } | **MINUTE** *[(precision)]* { **TO MINUTE** | **TO SECOND** | **TO FRACTION** [{ *(scale)* }] } | **SECOND** *[(precision)]* { **TO SECOND** | **TO FRACTION** } | **FRACTION TO FRACTION** [{ *(scale)* }] } | { **YEAR** *[(precision)]* { **TO YEAR** | **TO MONTH** } | **MONTH** *[(precision)]* **TO MONTH** } }"

Element	Description	Restrictions	Syntax
scale	Integer number of digits in FRACTION field. Default is 3.	Must be in the range from 1 to 5	Literal Number on page 1198
precision	Integer number of digits in the largest time unit that the INTERVAL includes. For YEAR, the default is 4. For all other time units except FRACTION, the default is 2.	Must be in the range from 1 to 9	Literal Number on page 1198

Usage

This segment specifies the precision and scale of an INTERVAL data type.

A keyword specifying the *largest* time unit must be the first keyword, and a keyword specifying the *smallest* time unit must follow the TO keyword. These can be the same keyword. This segment resembles the syntax of a #unique_162, but with these exceptions:

- If the largest time unit keyword is YEAR or MONTH, the smallest time unit keyword cannot specify a time unit smaller than MONTH.
- You can specify up to 9-digit *precision* after the first time unit, unless FRACTION is the first time unit (in which case no *precision* is valid after the first FRACTION keyword, but you can specify up to 5 digits of *scale* after the second FRACTION keyword).

Because *year* and *month* are not fixed-length units of time, the database server treats INTERVAL data types that include the YEAR or MONTH keywords in their qualifiers as incompatible with INTERVAL data types whose qualifiers are time units smaller than MONTH. The database server supports no implicit casts between these two categories of INTERVAL data types.

The next two examples show YEAR TO MONTH qualifiers of INTERVAL data types. The first example can hold an interval of up to 999 years and 11 months, because it gives `3` as the precision of the YEAR field. The second example uses the default precision on the YEAR field, so it can hold an interval of up to 9,999 years and 11 months.

```
YEAR (3) TO MONTH

YEAR TO MONTH
```

When you want a value to specify only one kind of time unit, the first and last qualifiers are the same. For example, an interval of whole years is qualified as YEAR TO YEAR or YEAR (5) TO YEAR, for an interval of up to 99,999 years.

The following examples show several forms of INTERVAL field qualifiers:

```
YEAR(5) TO MONTH

DAY (5) TO FRACTION(2)

DAY TO DAY

FRACTION TO FRACTION (4)
```

Example

For information about how to specify INTERVAL field qualifiers and how to use INTERVAL data in arithmetic and relational operations, see the related reference, INTERVAL data type.

Literal Collection

Use the Literal Collection segment to specify values for a collection data type. For the syntax of expressions that return values of individual elements within a collection, see Collection Constructors on page 1051.

Syntax

(explicit id unique_828_Connect_42_litco002) unique_828_Connect_42_litco002(explicit id unique_828_Connect_42_litco003) unique_828_Connect_42_litco003

Literal Collection



Literal Data



Usage

You can specify literal collection values for SET, MULTISET, or LIST data types.

To specify a single literal-collection value, specify the collection type and the literal values. The following SQL statement inserts four integer values into a column called **set_col** that was declared as `SET(INT NOT NULL)`:

```
INSERT INTO table1 (set_col) VALUES (SET{6, 9, 9, 4});
```

Specify an empty collection with an empty pair of braces ({ }) symbols. This example inserts an empty list into a column **list_col** that was declared as `LIST(INT NOT NULL):`

```
INSERT INTO table2 (list_col) VALUES ('LIST{}');
```

A pair of single (') or double (") quotation marks can delimit the collection. Double quotation marks are not valid, however, in databases where delimited identifiers are enabled, except to delimit SQL identifiers.

If you are passing a literal collection as an argument to an SPL routine, make sure that there is a blank space between the parentheses that surround the arguments and quotation marks that indicate the beginning and end of the literal collection.

Element Literal Value

The diagram for Literal Collection on page 1195 refers to this section. Elements of a collection can be literal values for the following data types.

For a Collection of Type	Literal Value Syntax
BOOLEAN	`t` or `f`, representing TRUE or FALSE as a quoted string
CHAR, VARCHAR, NCHAR, NVARCHAR, CHARACTER VARYING, LVARCHAR, DATE	Quoted String on page 1202
DATETIME	#unique_474
DECIMAL, MONEY, FLOAT, INTEGER, INT8, SMALLFLOAT, SMALLINT	Literal Number on page 1198

For a Collection of Type	Literal Value Syntax
INTERVAL	#unique_475
Opaque data types	Quoted String on page 1202. The string literal must be recognized by the input support function for the associated opaque type.
Row Type	Literal Row on page 1199. When the collection element type is a named ROW type, you do not need to cast the inserted values to the named ROW type.

Important: You cannot specify the simple-large-object data types (BYTE and TEXT) as the element type for a collection.

Quoted strings must be specified with a different type of quotation mark than the quotation marks that encompass the collection, so that the database server can parse the quoted strings. Thus, if you use double (") quotation marks to specify the collection, use single (') quotation marks to specify individual, quoted-string elements. (In databases where delimited identifiers are enabled, however, double quotation marks are not valid, except to delimit SQL identifiers.)

Nested Quotation Marks

The diagram for Literal Collection on page 1195 refers to this section.

A *nested collection* is a collection that is the element type for another collection.

Whenever you nest collection literals, use nested quotation marks. In these cases, you must follow the rule for nesting quotation marks. Otherwise, the database server cannot correctly parse the strings.

The general rule is that you must double the number of quotation marks for each new level of nesting. For example, if you use double (") quotation marks for the first level, you must use two double quotation marks for the second level, four double quotation marks for the third level, eight for the fourth level, sixteen for the fifth level, and so on.

Likewise, if you use single (') quotation marks for the first level, you must use two single quotation marks for the second level and four single quotation marks for the third level. There is no limit to the number of levels you can nest, as long as you follow this rule.

The following examples illustrate the case for two levels of nested collection literals, using double (") quotation marks. Here table **tab5** is a single-column table whose only column, **set_col**, is a nested collection type.

The following statement creates the **tab5** table:

```
CREATE TABLE tab5 (set_col SET(SET(INT NOT NULL) NOT NULL));
```

The following statement inserts values into the table **tab5**:

```
INSERT INTO tab5 VALUES ( "SET{""SET{34, 56, 23, 33}""}" );
```

For each literal value, the opening quotation mark and the closing quotation mark must match. Thus, if you open a literal with two double quotation marks, you must close that literal with two double quotation marks (`""a literal value""`).

To specify nested quotation marks within an SQL statement in programs, use the C escape character for every double quotation mark inside a string that is delimited by single quotation marks. Otherwise, the preprocessor cannot correctly interpret the literal collection value. For example, the preceding INSERT statement on the **tab5** table would appear in the program as follows:

```
EXEC SQL insert into tab5
   values ('set{\"set{34, 56, 23, 33}\"}');
```

For more information, see the chapter on complex data types in the *HCL® Informix® Enterprise Replication Guide*.

If the collection is a nested collection, you must include the collection-constructor syntax for each level of collection type. Suppose you define the following column:

```
nest_col SET(MULTISET (INT NOT NULL) NOT NULL);
```

The following statement inserts three elements into the **nest_col** column:

```
INSERT INTO tabx (nest_col)
   VALUES ("SET{'MULTISET{1, 2, 3}'}");
```

Literal Number

A *literal number* is the base-10 representation of a real number as an integer, as a fixed-point decimal number, or in exponential notation. Use the Literal Number segment whenever you see a reference to a literal number in a syntax diagram.

Syntax

Literal Number

"{ + | - }"

"{ *digit* [{ . | . *digit* }] | . *digit* }"

"[{ e | E }{ - | + } *digit*]"

Element	Description	Restrictions	Syntax
digit	Integer in range 0 through 9	Must be an ASCII digit	Literal entered from the keyboard.

Usage

You cannot include comma (`,`) or blank (ASCII 32) character. The unary plus (`+`) or minus (`-`) sign can precede a literal number, mantissa, or exponent.

You cannot include non-ASCII digits in literal numbers.

Integer Literals

An integer has no fractional part and cannot include a decimal point. Built-in data types of SQL that can be exactly represented as literal integers include BIGINT, BIGSERIAL, DECIMAL(*p*, 0), INT, INT8, SERIAL, SERIAL8, and SMALLINT.

If you use the representation of a number in a base other than 10 (such as a binary, octal, or hexadecimal) in any context where a literal integer is valid, the database server will attempt to interpret the value as a base-10 literal integer. For most data values, the result will be incorrect.

The following examples show some valid literal integers:

```
10            -27            +25567
```

Thousands separators (such as comma symbols) are not valid in literal integers, nor in any other literal number.

Fixed-Point Decimal Literals

Fixed-point decimal literals can exactly represent DECIMAL(*p*,*s*) and MONEY values. These can include a decimal point:

```
-123.456       00123456       +123456.0
```

The digits to the right of the decimal point in these examples are the fractional portions of the numbers.

Floating-Point Decimal Literals

Floating-point literals can exactly represent FLOAT, SMALLFLOAT, and DECIMAL(*p*) values, using a decimal point or exponential notation, or both. They can approximately represent real numbers in exponential notation. The next examples show floating point numbers:

```
-123.45E6      1.23456E2      123456.0E-3
```

The `E` in the previous examples is the symbol for exponential notation. The digit that follows `E` is the value of the exponent. For example, the number 3E5 (or 3E+5) means 3 multiplied by 10 to the fifth power, and the number 3E-5 means 3 multiplied by the reciprocal of 10 to the fifth power.

Literal Numbers and the MONEY Data Type

When you use a literal number as a MONEY value, do not include a currency symbol or include commas. The **DBMONEY** environment variable or the locale file can format how MONEY values are displayed in output.

Literal Row

The Literal Row segment specifies the syntax for literal values of named and unnamed ROW data types.

For expressions that evaluate to field values within a ROW data type, see ROW constructors on page 1050.

Syntax

(explicit id unique_829_Connect_42_liro002) unique_829_Connect_42_liro002(explicit id unique_829_Connect_42_liro003)

unique_829_Connect_42_liro003(explicit id unique_829_Connect_42_liro004) unique_829_Connect_42_liro004(explicit id unique_829_Connect_42_liro005)

unique_829_Connect_42_liro005(explicit id unique_829_Connect_42_liro006) unique_829_Connect_42_liro006

Literal Row

```
" { ' ROW( { <Field Literal Value> } ) ' | ROW( { <Field Literal Value> } ) } "
```

Field Literal Value

```
" { <Quoted String>[] | <Literal Number>[] | USER | <Literal DATETIME>[] | <Literal INTERVAL>[] | <Literal
Collection>[] | literal_opaque_type | 'literal_BOOLEAN' | ROW( <Literal Row> ) } "
```

Element	Description	Restrictions	Syntax
literal_opaque_type	Literal representation for an opaque data type	Must be a literal that is recognized by the input support function for the associated opaque data type	Defined by the developer of the opaque data type
literal_BOOLEAN	Literal representation of a BOOLEAN value	Must be either `'t'` (= TRUE) or `'f'` (= FALSE) specified as a quoted string	Quoted String on page 1202

Usage

You can specify literal values for named ROW and unnamed ROW data types. A ROW constructor introduces a literal ROW value, which can optionally be enclosed between quotation marks.

The format of the value for each field of the ROW type must be compatible with the data type of the corresponding field.

Important: You cannot specify simple-large-object data types (BYTE or TEXT) as the field type for a row.

Fields of a row can be literal values for the data types in the following table.

For a Field of Type	Literal Value Syntax
BOOLEAN	`t` or `f`, representing TRUE or FALSE
CHAR, VARCHAR, LVARCHAR, NCHAR, NVARCHAR, CHARACTER VARYING, DATE	Quoted String on page 1202
DATETIME	#unique_474

For a Field of Type	Literal Value Syntax
BIGINT, DECIMAL, MONEY, FLOAT, INTEGER, INT8, SMALLFLOAT, SMALLINT	Literal Number on page 1198
INTERVAL	#unique_475
Opaque data types	Quoted String on page 1202 The string must be a literal that is recognized by the input support function for the associated opaque type.
Collection type (SET, MULTISET, LIST)	Literal Collection on page 1195 For information on literal collection values as variable or column values, see Nested Quotation Marks on page 1197. For information on literal collection values for a ROW type, see Literals for Nested Rows on page 1202.
Another ROW type (named or unnamed)	For information on ROW type values, see Literals for Nested Rows on page 1202.

Literals of an Unnamed Row Type

To specify a literal value for an unnamed ROW type, introduce the literal row with the ROW constructor; you must enclose the values between parentheses. For example, suppose that you define the **rectangles** table as follows:

```
CREATE TABLE rectangles
(
   area FLOAT,
   rect ROW(x INTEGER, y INTEGER, length FLOAT, width FLOAT),
)
```

The following INSERT statement inserts values into the **rect** column of the **rectangles** table:

```
INSERT INTO rectangles (rect)
   VALUES ("ROW(7, 3, 6.0, 2.0)")
```

Literals of a Named Row Type

To specify a literal value for a named ROW type, introduce the literal row with the ROW type constructor and enclose the literal values for each field in parentheses. In addition, you can cast the row literal to the appropriate named ROW type to ensure that the row value is generated as a named ROW type. The following statements create the named ROW type **address_t** and the **employee** table:

```
CREATE ROW TYPE address_t
(
street CHAR(20),
city CHAR(15),
state CHAR(2),
zipcode CHAR(9)
```

```
);

CREATE TABLE employee
(
   name CHAR(30),
   address address_t
);
```

The following INSERT statement inserts values into the **address** column of the **employee** table:

```
INSERT INTO employee (address)
VALUES (
"ROW('103 Baker St', 'Tracy','CA', 94060)"::address_t)
```

Literals for Nested Rows

If the literal value is for a nested row, specify the ROW type constructor for each row level. If you include quotation marks as delimiters, they should enclose the outermost row. For example, suppose that you create the **emp_tab** table:

```
CREATE TABLE emp_tab
(
   emp_name CHAR(10),
   emp_info ROW( stats ROW(x INT, y INT, z FLOAT))
);
```

The following INSERT statement adds a row to the **emp_tab** table:

```
INSERT INTO emp_tab VALUES ('joe boyd', "ROW(ROW(8,1,12.0))" );
```

Similarly, if the row-string literal contains a nested collection, only the outermost literal row can be enclosed between quotation marks. Do not put quotation marks around an inner, nested collection type.

Quoted String

A quoted string is a string literal between quotation marks. Use this segment whenever you see a reference to a quoted string in a syntax diagram.

Syntax

(explicit id unique_36_Connect_42_quost002) unique_36_Connect_42_quost002

Quoted String

" { ' [{ | *character* | '' }] ' | " [{ | *character* | "" }] " }

Element	Description	Restrictions	Syntax
character	Code set element within quoted string	Cannot enclose between double quotation marks if the **DELIMIDENT** environment variable is set	Literal value from the keyboard

Usage

Use quoted strings to specify string literals in data-manipulation statements and other SQL statements. For example, you can use a quoted string in an INSERT statement to insert a value into a column of a character data type.

Restrictions on Specifying Characters in Quoted Strings

You must observe the following restrictions on *character* in quoted strings:

- If you are using the ASCII code set, you can specify any printable ASCII character, including a single (') quotation mark or double (") quotation mark. For restrictions that apply to using quotation marks in quoted strings, see Using Quotation Marks in Strings on page 1205.
- In some locales, you can specify non-ASCII characters, including multibyte characters, that the locale supports. See the discussion of quoted strings in the *HCL® Informix® GLS User's Guide*.
- If you enable newline characters for quoted strings, you can embed newline characters in quoted strings. For further information, see Newline Characters in Quoted Strings on page 1204.
- You can enter DATETIME and INTERVAL data values as quoted strings. For the restrictions that apply to entering DATETIME and INTERVAL data in quoted-string format, see DATETIME and INTERVAL Values as Strings on page 1205.
- Quoted strings that are used with the LIKE or MATCHES keyword in a search condition can include wildcard characters that have a special meaning in the search condition. For further information, see LIKE and MATCHES in a Condition on page 1205.
- When you insert a value that is a quoted string, you must observe a number of restrictions. For further information, see Inserting Values as Quoted Strings on page 1205.

The DELIMIDENT Environment Variable

If the **DELIMIDENT** environment variable is set on the database server, you cannot use double quotation marks (") to delimit literal strings. If **DELIMIDENT** is set, the database server interprets strings enclosed in double quotation marks as SQL identifiers, not as literal strings. If **DELIMIDENT** is not set, a string between double quotation marks is interpreted as a literal string, not an identifier. For further information, see Using Quotation Marks in Strings on page 1205, and the description of **DELIMIDENT** in *HCL® Informix® Guide to SQL: Reference*.

DELIMIDENT is also supported on client systems, where it can be set to `y`, to `n`, or to no setting.

- `y` specifies that client applications must use single quotation mark (') symbols to delimit literal strings, and must use double quotation mark (") symbols only around delimited SQL identifiers. Delimited identifiers can support a larger character set than is valid for undelimited identifiers. Letters within delimited strings or delimited identifiers are case-sensitive.
- `n` specifies that client applications can use double quotation mark (") or single quotation mark (') symbols to delimit character strings, but not to delimit SQL identifiers. If the database server encounters a string delimited by double or single quotation mark symbols in a context where an SQL identifier is required, it issues an error. An owner name,

however, that qualifies an SQL identifier can be delimited by single quotation mark (') symbols. You must use a pair of the same quotation mark symbols to delimit a character string.

- Specifying **DELIMIDENT** with no value on the client system requires client applications to use the **DELIMIDENT** setting that is the default for their application programming interface (API).

Client APIs of Informix® use the following default **DELIMIDENT** settings:

- For OLE DB and .NET, the default **DELIMIDENT** setting is `y`
- For ESQL/C, JDBC, and ODBC, the default **DELIMIDENT** setting is `n`
- APIs that have ESQL/C as an underlying layer, the (LIBDMI), and the C++ API, behave as ESQL/C, and use 'n' as the default if no value for **DELIMIDENT** is specified on the client system.

Even if **DELIMIDENT** is set, you can use single quotation mark (') symbols to delimit *authorization identifiers* as the owner name component of a database object name, as in the following example:

```
RENAME COLUMN 'Owner'.table2.collum3 TO column3;
```

The general rule, however, is that when **DELIMIDENT** is set, the SQL parser interprets strings delimited by single quotation marks as string literals, and interprets character strings delimited by double quotation marks (") as SQL identifiers.

Newline Characters in Quoted Strings

By default, the string constant must be written on a single line. That is, you cannot use embedded newline characters in a quoted string. You can, however, override this default behavior in one of two ways:

- To enable newline characters in quoted strings in all sessions, set the ALLOW_NEWLINE parameter to 1 in the `ONCONFIG` file.
- To enable newline characters in quoted strings for the current session, execute the built-in function **IFX_ALLOW_NEWLINE**.

This enables newline characters in quoted strings for the current session:

```
EXECUTE PROCEDURE IFX_ALLOW_NEWLINE('T');
```

If newline characters in quoted strings are not enabled for a session, the following statement is invalid and returns an error:

```
SELECT 'The quick brown fox
   jumped over the old gray fence'
   FROM customer
   WHERE customer_num = 101;
```

If you enable newline characters in quoted strings for the session, however, the statement in the preceding example is valid and executes successfully.

For more information on the **IFX_ALLOW_NEWLINE** function, see IFX_ALLOW_NEWLINE Function on page 1148. For more information on the ALLOW_NEWLINE parameter in the `ONCONFIG` file, see your *HCL® Informix® Administrator's Reference*.

Using Quotation Marks in Strings

The single quotation mark (') has no special significance in string literals delimited by double quotation marks. Conversely, double quotation mark (") has no special significance in strings delimited by single quotation marks. For example, these strings are valid:

```
"Nancy's puppy jumped the fence"
'Billy told his kitten, "No!" '
```

A string delimited by double quotation marks can include a double quotation mark character by preceding it with another double quotation mark, as the following string shows:

```
"Enter ""y"" to select this row"
```

When the **DELIMIDENT** environment variable is set, double quotation marks can only delimit SQL identifiers, not strings. For more information on delimited identifiers, see Delimited Identifiers on page 1233.

DATETIME and INTERVAL Values as Strings

You can enter DATETIME and INTERVAL data in the literal forms described in the #unique_474 and #unique_475, or you can enter them as quoted strings.

Valid literals that are entered as character strings are converted automatically into DATETIME or INTERVAL values.

These statements enter INTERVAL and DATETIME values as quoted strings:

```
INSERT INTO cust_calls(call_dtime) VALUES ('2007-5-4 10:12:11');
INSERT INTO manufact(lead_time) VALUES ('14');
```

The format of the value in the quoted string must exactly match the format specified by the INTERVAL or DATETIME qualifiers of the column. For the first INSERT in the preceding example, the **call_dtime** column must be defined with the qualifiers YEAR TO SECOND for the INSERT statement to be valid.

LIKE and MATCHES in a Condition

Quoted strings with the LIKE or MATCHES keyword in a condition can include wildcard characters. For a complete description of how to use wildcard characters, see Condition on page 967.

Inserting Values as Quoted Strings

In the default locale, if you are inserting a value that is a quoted string, you must adhere to the following restrictions:

- Enclose CHAR, VARCHAR, NCHAR, NVARCHAR, DATE, DATETIME, INTERVAL, and LVARCHAR values in quotation marks.
- Specify DATE values in the *mm/dd/yyyy* format (or in the format that the **DBDATE** or **GL_DATE** environment variable specifies, if set).
- You cannot insert strings longer than 32 kilobytes.
- Numbers with decimal values must include a decimal separator. Comma (,) is not valid as a decimal separator in the default locale.

- MONEY values cannot include a dollar sign ($) or commas.
- You can enter NULL in a column only if it accepts null values.

Numeric Operations on Character Columns

Avoid comparing number literals to character columns. It requires that all of the strings compared be converted to numbers, which takes much longer than comparing two strings.

For example, suppose that you wish to find all customers within the 356 telephone exchange code:

```
SELECT lname FROM customer WHERE phone [5,7] = '356';
```

Notice that the operand whose value is 356 is enclosed in quotes. The quotes indicate that the database server must handle the filter as a character string. By contrast, when the operand is not in quotes, the server treats each retrieved value as a number, and must implicitly cast each value retrieved from the table to a numeric data type.

The following example causes implicit data type conversion of the **phone** substrings:

```
SELECT lname FROM customer WHERE phone [5,7] = 356;
```

If the UPDATE STATISTICS MEDIUM or UPDATE STATISTICS HIGH statement has been run on this column, the query optimizer tries to determine the selectivity of the predicate by matching the constant in the query with a substring of values saved in the distribution bin. Requiring data type conversion of every row in a character column so that it can be compared to a numeric filter needlessly increases the cost of the query that omits quotation mark delimiters around `356`, compared to cost of the query in the first example.

Queries that compare character strings to numbers can fail with EM -1213 if the database server cannot convert the string. If you cannot avoid applying numeric filters to character values, only attempt such operations on character columns whose characters are restricted to digits in the range ASCII 0x30 through 0x39, and decimal point (ASCII 0x2e). This range is also known as *seminumeric*.

The database server does not use an index when DML statements compare a character column with a noncharacter value that is not equal in length to the character column.

Relational Operator

A relational operator compares two expressions quantitatively. Use the Relational Operator segment whenever you see a reference to a relational operator in a syntax diagram.

Syntax

(explicit id unique_1039_Connect_42_relop001) unique_1039_Connect_42_relop001

Relational Operator

"{ < | <= | > |{ = | == }| >= | <> |[]!= }"

Usage

The relational operators of SQL have the following meanings.

Relational Operator	Meaning
<	Less than
<=	Less than or equal to
>	Greater than
= *or* ==	Equal to
>=	Greater than or equal to
<> *or* !=	Not equal to

Usage

For number expressions, *greater than* means to the right on the real line.

For DATE and DATETIME expressions, *greater than* means later in time.

For INTERVAL expressions, *greater than* means a longer span of time.

For CHAR, VARCHAR, and LVARCHAR expressions, *greater than* means *after* in code-set order.

For NCHAR and NVARCHAR expressions, *greater than* means *after* in the localized collation order, if one exists; otherwise, *greater than* means *after* in code-set order.

Locale-based collation order, if one is defined for the locale, is used for NCHAR and NVARCHAR expressions. So for NCHAR and NVARCHAR expressions, *greater than* means *after* in the locale-based collation order. For more information on locale-based collation order and the NCHAR and NVARCHAR data types, see the *HCL® Informix® GLS User's Guide*.

For information on how relational operator expressions with NCHAR and NVARCHAR operands in databases that have the NLCASE INSENSITIVE property differ from their behavior in databases that are case sensitive, see the topic NCHAR and NVARCHAR expressions in case-insensitive databases.

Using Operator Functions in Place of Relational Operators

Each relational operator is bound to a particular operator function, as the table shows. The operator function accepts two values and returns a boolean value of true, false, or unknown.

Relational Operator	Associated Operator Function
<	**lessthan( )**
<=	**lessthanorequal( )**
>	**greaterthan( )**
>=	**greaterthanorequal( )**
= *or* ==	**equal( )**
<> *or* !=	**notequal( )**

Connecting two expressions with a relational operator is equivalent to invoking the operator function on the expressions. For example, the next two statements both select orders with a shipping charge of $18.00 or more.

The `>=` operator in the first statement implicitly invokes the **greaterthanorequal( )** operator function:

```
SELECT order_num FROM orders
   WHERE ship_charge >= 18.00;

SELECT order_num FROM orders
   WHERE greaterthanorequal(ship_charge, 18.00);
```

The database server provides the operator functions associated with the relational operators for all built-in data types. When you develop a user-defined data type, you must define the operator functions for that type for users to be able to use the relational operator on the type.

If you define **lessthan( )**, **greaterthan( )**, and the other operator functions for a user-defined type, then you should also define **compare( )**. Similarly, if you define **compare( )**, then you should also define **lessthan( )**, **greaterthan( )**, and the other operator functions. All of these functions must be defined in a consistent manner, to avoid the possibility of incorrect query results when UDT values are compared in the WHERE clause of a SELECT.

Support for ASCII Characters in Nondefault Code Sets (GLS)

Most code sets for nondefault locales (called *nondefault code sets*) support the ASCII characters. In a nondefault locale, the database server uses ASCII code-set order for ASCII data in CHAR and VARCHAR expressions, if the code set supports these ASCII characters. If the current collation (as specified by **DB_LOCALE** or by SET COLLATION) supports a localized collating order, however, that localized order is used when the database server sorts NCHAR or NVARCHAR values.

Literal Numbers as Operands

You might obtain unexpected results if a literal number that you specify as an operand is not in a format that can exactly represent the data type of another value with which it is compared by a relational operator. Because of rounding errors, for example, a relational operator like `=` or the **equals( )** operator function generally cannot return `TRUE` if one operand returns a FLOAT value and the other an INTEGER. For information about which of the built-in data types store values that can be exactly represented as literal numbers, see the section Literal Number on page 1198.

Other syntax segments

These topics describe *syntax segments*, which are language elements, such as database object names or optimizer directives, that appear as a subdiagram reference in the syntax diagrams of some SQL or SPL statements.

Most segments that can occur in only one statement are described in SQL statements on page 18 or SPL statements on page 911 within the description of the statement. For the sake of clarity, ease of use, and comprehensive treatment, however, most segments that can occur in various SQL or SPL statements, and that are not data types nor expressions, are discussed separately here.

The previous chapter described the syntax segments that specify data types and expressions. This chapter describes additional syntax segments that are neither data types, expressions, nor complete SQL statements or SPL statements. These segments are referenced in various syntax diagrams that appear in SQL statements on page 18 and in other chapters of this document.

Arguments

Use the Argument segment to pass a specific value as input to a routine. Use this segment wherever you see a reference to an *argument* in a syntax diagram.

Syntax

(explicit id unique_702_Connect_42_arg101) unique_702_Connect_42_arg101

Argument

" [*parameter*=] "

" { <Subset of Expression> [] | **NULL** | (*singleton_select*) } "

Element	Description	Restrictions	Syntax
parameter	A parameter whose value you specify	Must match a name that CREATE FUNCTION or CREATE PROCEDURE statement declared	Identifier on page 1231
singleton_select	Embedded query that returns a single value	Must return exactly one value of a data type and length compatible with *parameter*	SELECT statement on page 690

Usage

The CREATE PROCEDURE or CREATE FUNCTION statement can define a parameter list for a UDR. If the parameter list is not empty, you must enter arguments when you invoke the UDR. An *argument* is a specific value whose data type is compatible with that of the corresponding UDR parameter.

When you execute a UDR, you can enter arguments in either of two ways:

- With a parameter name (in the form *parameter name = expression)*, even if the arguments are not in the same order as the parameters
- By position, with no *parameter* name, where each *expression* is in the same order as the parameter to which the argument corresponds. (This is sometimes called *ordinal* format.)

You cannot mix these two ways of specifying arguments within a single invocation of a routine. If you specify a *parameter* name for one argument, for example, you must use parameter names for all the arguments.

In the following example, both statements are valid for a user-defined procedure that expects three character arguments, **t**, **d**, and **n**:

```
EXECUTE PROCEDURE add_col (t ='customer', d ='integer',
   n ='newint');

EXECUTE PROCEDURE add_col ('customer','newint','integer') ;
```

Comparing Arguments to the Parameter List

When you create or register a UDR with CREATE PROCEDURE or CREATE FUNCTION, you declare a *parameter list* with the names and data types of the parameters that the UDR expects. (Parameter names are optional for external routines written in the C or Java™ languages.) See Routine Parameter List on page 1281 for details of declaring parameters.

User-defined routines can be *overloaded*, if different routines have the same identifier, but have different numbers of declared parameters. For more information about overloading, see Routine Overloading and Routine Signatures on page 1228.

If you attempt to execute a UDR with more arguments than the UDR expects, you receive an error.

If you invoke a UDR with fewer arguments than the UDR expects, the omitted arguments are said to be *missing*. The database server initializes missing arguments to their corresponding default values. This initialization occurs before the first executable statement in the body of the UDR.

If missing arguments have no default values, Informix® issues an error.

Named parameters cannot be used to invoke UDRs that overload data types in their routine signatures. Named parameters are valid in resolving non-unique routine names only if the signatures have different numbers of parameters:

```
func( x::integer, y );    -- VALID if only these 2 routines
func( x::integer, y, z ); -- have the same 'func' identifier

func( x::integer, y );    -- NOT VALID if both routines have
func( x::float, y ;       -- same identifier and 2 parameters
```

For both ordinal and named parameters, the routine with the fewest parameters is executed if two or more UDR signatures have multiple numbers of defaults:

```
func( x, y default 1 )
func( x, y default 1, z default 2 )
```

If two registered UDRs that are both called **func** have the signatures shown above, then the statement `EXECUTE func(100)` invokes **func(100,1)**.

You cannot supply a subset of default values using named parameters unless they are in the positional order of the routine signature. That is, you cannot skip a few arguments and rely on the database server to supply their default values.

For example, given the signature:

```
func( x, y default 1, z default 2 )
```

you can execute:

```
func( x=1, y=3 )
```

but you cannot execute:

```
func( x=1, z=3 )
```

Subset of Expressions Valid as an Argument

The diagram for Arguments on page 1209 refers to this section.

You can use any expression as an argument, except an aggregate function. If you use a subquery or function call as an argument, the subquery or function must return a single value of the appropriate data type and size. For the syntax and usage of SQL expressions, see Expression on page 1003.

Arguments to UDRs in Remote Databases

UDRs are valid in cross-database and in cross-server distributed operations in most contexts where a UDR is valid in the local database, but every participating database must have the same logging mode.

Excluding BIGSERIAL, BYTE, SERIAL, SERIAL8, and TEXT, the data types that are valid as arguments to cross-server UDRs include the built-in SQL data types that are not opaque, as listed in Data Types in Distributed Queries on page 700, and these additional built-in opaque and DISTINCT data types:

- BOOLEAN
- LVARCHAR
- DISTINCT of built-in types that are not opaque

- DISTINCT of BOOLEAN
- DISTINCT of LVARCHAR
- DISTINCT of the DISTINCT types listed above.

These data types can be arguments to SPL, C, or Java™ language UDRs, if the UDRs are defined in all the participating databases. Any implicit or explicit casts defined over these data types must be duplicated across all the participating Informix® instances. The DISTINCT data types must have exactly the same data type hierarchy defined in all databases that participate in the distributed query.

The same data types are valid as arguments in calls to UDRs in other databases of the same Informix® instance, as well as arguments of the following additional types:

- BLOB
- CLOB
- UDTs that you cast explicitly to built-in types

All the UDRs, UDTs, DISTINCT data types, DISTINCT type hierarchies, casts, and cast functions must be registered in all of the participating databases. For more information on DISTINCT types in distributed operations, see DISTINCT Types in Distributed Operations on page 996.

Collection-Derived Table

A *collection-derived table* is a virtual table in which the values in the rows of the table are equivalent to elements of a collection. Use this segment where you see a reference to Collection-Derived Table in a syntax diagram. This syntax is an extension to the ANSI/ISO standard for SQL.

Syntax

(explicit id unique_614_Connect_42_codt002) unique_614_Connect_42_codt002(explicit id unique_614_Connect_42_codt003)

unique_614_Connect_42_codt003(explicit id unique_614_Connect_42_codt004) unique_614_Connect_42_codt004

Collection-Derived Table

" TABLE ("

" { *collection_expr*) [{ AS *alias* | *alias* }] [(*derived_column*)] | { *collection_var* | *row_var* }) } "

Element	Description	Restrictions	Syntax
alias	Temporary name for a collection-derived table whose scope is a SELECT statement. The default is implementation dependent.	If potentially ambiguous, you must precede *alias* with the AS keyword. See Aliases for Tables or Views on page 713.	Identifier on page 1231

Element	Description	Restrictions	Syntax
collection_expr	Any expression that evaluates to the elements of a single collection	See Restrictions with the Collection-Expression Format on page 1214.	Expression on page 1003
collection_var, *row_var*	Name of a typed or untyped collection variable, or **row** variable that holds the collection-derived table	Must have been declared in the program or (for *collection_var*) in an SPL routine	See the *HCL® Informix® Enterprise Replication Guide* or DEFINE on page 920.
derived _column	Temporary name for a derived column in a table	If the underlying collection is not of a ROW data type, you can specify only one derived-column name	Identifier on page 1231

Usage

A collection-derived table can appear where a *table* name is valid in the UPDATE statement, in the FROM clause of the SELECT or DELETE statement, or in the INTO clause of an INSERT statement.

Use the collection-derived-table segment to accomplish these tasks:

- Access the elements of a collection as you would the rows of a table.
- Specify a collection variable to access, instead of a table name.
- Specify an ESQL/C **row** variable to access, instead of a table name.

The TABLE keyword converts a collection into a virtual table. You can use the collection expression format to query a collection column, or you can use the **collection** variable or **row** variable format to manipulate the data in a collection column.

Accessing a Collection Through a Virtual Table

When you use the collection expression format of the collection-derived table segment to access the elements of a collection, you can select elements of the collection directly through a virtual table. You can use this format in the FROM clause of a SELECT statement. The FROM clause can be in either a query or a subquery.

With this format you can use joins, aggregates, the WHERE clause, expressions, the ORDER BY clause, and other operations that are not available when you use the collection-variable format. This format reduces the need for multiple cursors and temporary tables.

Examples of possible collection expressions include column references, scalar subquery, dotted expression, functions, operators (through overloading), collection subqueries, literal collections, collection constructors, cast functions, and so on.

The following example uses a SELECT statement in the FROM clause whose result set defines a virtual table consisting of the fifty-first through seventieth qualifying rows, ordered by the **employee_id** column value.

```
SELECT * FROM TABLE(MULTISET(SELECT SKIP 50 FIRST 20 * FROM employees
   ORDER BY employee_id)) vt(x,y), tab2 WHERE tab2.id = vt.x;
```

The following example uses a join query to create a virtual table of no more than twenty rows (beginning with the 41st row), ordered by value in the **salary** column of the collection-derived table:

```
SELECT emp_id, emp_name, emp_salary
   FROM  TABLE(MULTISET(SELECT SKIP 40 LIMIT 20 id, name, salary
                        FROM e1, e2
                        WHERE e1.id = e2.id ORDER BY salary ))
   AS etab(emp_id, emp_name, emp_salary);
```

Table Expressions in the FROM Clause

Informix® supports ANSI/ISO standard syntax for table expressions in the FROM clause of SELECT queries and subqueries as a substitute for the Informix®-extension collection-derived table syntax. The keywords TABLE and MULTISET were required in version 10.00 and in earlier releases. These extensions to the ANSI/ISO standard for SQL are supported but no longer required for collection-derived table specifications in the FROM clause of SELECT statements.

The following two queries return the same result set, but only the second query complies with the ANSI/ISO standard:

```
SELECT * FROM TABLE(MULTISET(SELECT col1 FROM tab1 WHERE col1 = 100))
   AS vtab(c1),
   (SELECT col1 FROM tab1 WHERE col1 = 10) AS vtab1(vc1) ORDER BY c1;

SELECT * FROM (SELECT col1 FROM tab1 WHERE col1 = 100) AS vtab(c1),
   (SELECT col1 FROM tab1 WHERE col1 = 10) AS vtab1(vc1)
      ORDER BY c1;
```

The same SELECT statement can combine instances of both the Informix®-extension and ANSI/ISO syntax for derived tables:

```
SELECT * FROM (SELECT col1 FROM tab1 WHERE col1 = 100) AS vtab(c1),
   TABLE(MULTISET(SELECT col1 FROM tab1 WHERE col1 = 10)) AS vtab1(vc1)
      ORDER BY c1;
```

The subquery must be delimited by parentheses in both formats, but the outer set of parentheses (( )) that immediately follows the TABLE keyword and encloses the MULTISET collection subquery specification is an extension to the ANSI/ISO syntax. This ANSI/ISO syntax is valid only in the FROM clause of the SELECT statement. You cannot omit these keywords and parentheses from a collection subquery specification in any other context.

Restrictions with the Collection-Expression Format

When you use the collection-expression format, certain restrictions apply:

- A collection-derived table is read-only.
  - It cannot be the target of INSERT, UPDATE, or DELETE statements.

    To perform insert, update, and delete operations, you must use the collection-variable format.
  - It cannot be the underlying table of an updatable cursor or view.

- In the FROM clause of the SELECT statement, the CALL keyword of SPL cannot precede the TABLE keyword of a table expression.
- If the collection is a LIST data type, the resulting collection-derived table does not preserve the order of the elements in the LIST.
- The underlying collection expression cannot evaluate to NULL.
- The collection expression cannot contain a reference to a collection on a remote database server.
- The collection expression cannot contain column references to tables that appear in the same FROM clause. That is, the collection-derived table must be independent of other tables in the FROM clause.

  For example, the following statement returns an error because the collection-derived table, `TABLE (parents.children),` refers to the **parents** table, which is also referenced in the FROM clause:

  ```
  SELECT COUNT(*)
     FROM parents, TABLE(parents.children) c_table
     WHERE parents.id = 1001;
  ```

  To counter this restriction, you might write a query that contains a subquery in the Projection clause:

  ```
  SELECT (SELECT COUNT(*)
        FROM TABLE(parents.children) c_table)
     FROM parents WHERE parents.id = 1001;
  ```

Additional Restrictions That Apply to ESQL/C

In addition to the previously described restrictions, the following restrictions also apply when you use the collection-expression format with :

- You cannot specify an untyped COLLECTION as the host-variable data type.
- You cannot use the format `TABLE(?).`

  The data type of the underlying collection variable must be determined statically. To counter this restriction, you can explicitly cast the variable to a typed collection data type (SET, MULTISET, or LIST) that the database server recognizes. For example,

  ```
  TABLE(CAST(? AS type))
  ```

- You cannot use the format `TABLE(:`*hostvar*`).`

  To counter this restriction, you must explicitly cast the variable to a typed collection data type (SET, MULTISET, or LIST) that the database server recognizes. For example,

  ```
  TABLE(CAST(:hostvar AS type))
  ```

Row Type of the Resulting Collection-Derived Table

If you do not specify a derived-column name, the behavior of the database server depends on the data types of the elements in the underlying collection.

Although a collection-derived table appears to contain columns of individual data types, these columns are, in fact, the fields of a ROW data type. The data type of the ROW type as well as the column name depend on several factors.

If the data type of the elements of the underlying collection expression is *type*, the database server determines the ROW type of the collection-derived table by the following rules:

- If *type* is a ROW data type, and no derived-column list is specified, then the ROW type of the collection-derived table is *type*.
- If *type* is a ROW data type and a derived column list is specified, then the ROW type of the collection-derived table is an unnamed ROW type whose column data types are the same as those of *type* and whose column names are taken from the derived column list.
- If *type* is not a ROW data type, the ROW type of the collection-derived table is an unnamed ROW type that contains one column of *type* and whose name is specified in the derived column list. If no name is specified, the database server assigns an implementation-dependent name to the column.

The extended examples that the following table shows illustrate these rules. The table uses the following schema for its examples:

```
CREATE ROW TYPE person (name CHAR(255), id INT);
CREATE TABLE parents
   (
   name CHAR(255),
   id INT,
   children LIST (person NOT NULL)
   );
CREATE TABLE parents2
   (
   name CHAR(255),
   id INT,
   children_ids LIST (INT NOT NULL)
   );
```

ROW Type	Explicit Derived-Column List	Resulting ROW Type of the Collection-Derived Table	Code Example
Yes	No	*Type*	`SELECT (SELECT c_table.name FROM TABLE(parents.children) c_table WHERE c_table.id = 1002) FROM parents WHERE parents.id = 1001;` In this example, the ROW type of **c_table** is **parents**.
Yes	Yes	Unnamed ROW type of which the column type is *Type* and the column name is the name in the derived-column list	`SELECT (SELECT c_table.c_name FROM TABLE(parents.children) c_table(c._name, c_id) WHERE c_table.c_id = 1002) FROM parents WHERE parents.id = 1001;` In this example, the ROW type of **c_table** is ROW(**c_name** CHAR(255), **c_id** INT).

ROW Type	Explicit Derived-Column List	Resulting ROW Type of the Collection-Derived Table	Code Example
No	No	Unnamed ROW that contains one column of *Type* that is assigned an implementation-dependent name	In the following example, if you do not specify **c_id**, the database server assigns a name to the derived column. In this case, the ROW type of **c_table** is ROW(*server_defined_name* INT).
No	Yes	Unnamed ROW type that contains one column of *Type* whose name is in the derived-column list	`SELECT(SELECT c_table.c_id FROM TABLE(parents2.child_ids) c_table (c_id) WHERE c_table.c_id = 1002) FROM parents WHERE parents.id = 1001`; Here the ROW type of **c_table** is ROW(**c_id** INT).

The following program fragment creates a collection-derived table using an SPL function that returns a single value:

```
CREATE TABLE wanted(person_id int);
CREATE FUNCTION
    wanted_person_count (person_set SET(person NOT NULL))
RETURNS INT;
RETURN( SELECT COUNT (*)
    FROM TABLE (person_set) c_table, wanted
    WHERE c_tabel.id = wanted.person_id);
END FUNCTION;
```

The next program fragment shows the more general case of creating a collection-derived table using an SPL function that returns multiple values:

```
-- Table of categories and child categories,
-- allowing any number of levels of subcategories
CREATE TABLE CategoryChild (
         categoryId      INTEGER,
         childCategoryId SMALLINT
);

INSERT INTO CategoryChild VALUES (1, 2);
INSERT INTO CategoryChild VALUES (1, 3);
INSERT INTO CategoryChild VALUES (1, 4);
INSERT INTO CategoryChild VALUES (2, 5);
INSERT INTO CategoryChild VALUES (2, 6);
INSERT INTO CategoryChild VALUES (5, 7);
INSERT INTO CategoryChild VALUES (7, 8);
INSERT INTO CategoryChild VALUES (7, 9);
INSERT INTO CategoryChild VALUES (4, 10);

-- "R" == ROW type
CREATE ROW TYPE categoryLevelR (
         categoryId      INTEGER,
         level   SMALLINT );
```

```
-- DROP FUNCTION categoryDescendants (
--               INTEGER, SMALLINT );
CREATE FUNCTION categoryDescendants (
         pCategoryId INTEGER,
         pLevel      SMALLINT DEFAULT 0 )
RETURNS MULTISET (categoryLevelR NOT NULL)

-- "p" == Prefix for Parameter names
-- "l" == Prefix for Local variable names
DEFINE lCategoryId LIKE CategoryChild.categoryId;
DEFINE lRetSet MULTISET (categoryLevelR NOT NULL);
DEFINE lCatRow categoryLevelR;

-- TRACE ON;
-- Must initialize collection before inserting rows
LET lRetSet = 'MULTISET{}' :: MULTISET (categoryLevelR NOT NULL);
```

```
FOREACH
SELECT childCategoryId INTO lCategoryId
   FROM CategoryChild WHERE categoryId = pCategoryId;
INSERT INTO TABLE (lRetSet)
   VALUES (ROW (lCategoryId, pLevel+1)::categoryLevelR);

-- INSERT INTO TABLE (lRetSet);
-- EXECUTE FUNCTION categoryDescendantsR ( lCategoryId,
-- pLevel+1 );
-- Need to iterate over results and insert into SET.
-- See the SQL Tutorial, pg. 10-52:
-- "Tip: You can only insert one value at a time
-- into a simple collection."
   FOREACH
   EXECUTE FUNCTION categoryDescendantsR ( lCategoryId, pLevel+1 )
     INTO lCatRow;
       INSERT INTO TABLE (lRetSet)
           VALUES (lCatRow);
   END FOREACH;
END FOREACH;

RETURN lRetSet;
END FUNCTION
;
-- "R" == recursive
-- DROP FUNCTION categoryDescendantsR (INTEGER, SMALLINT);
CREATE FUNCTION categoryDescendantsR (
         pCategoryId INTEGER,
         pLevel      SMALLINT DEFAULT 0
)
RETURNS categoryLevelR;
DEFINE lCategoryId      LIKE CategoryChild.categoryId;
DEFINE lCatRow          categoryLevelR;

FOREACH
   SELECT  childCategoryId
   INTO    lCategoryId
   FROM    CategoryChild
   WHERE   categoryId = pCategoryId
   RETURN ROW (lCategoryId, pLevel+1)::categoryLevelR WITH RESUME;
```

```
   FOREACH
   EXECUTE FUNCTION categoryDescendantsR ( lCategoryId, pLevel+1 )
      INTO    lCatRow
      RETURN  lCatRow WITH RESUME;
   END FOREACH;
END FOREACH;
END FUNCTION;

-- Test the functions:
SELECT lev, col
FROM    TABLE ((
         categoryDescendants (1, 0)
         )) AS CD (col, lev);
```

Accessing a Collection Through a Collection Variable

When you use the collection-variable format of the collection-derived table segment, you use a host or program variable to access and manipulate the elements of a collection. This format allows you to modify the contents of a variable as you would a table in the database, and then update the actual table with the contents of the **collection** variable.

You can use the collection-variable format (the TABLE keyword preceding a **collection** variable) in place of the name of a table, synonym, or view in the following SQL statements (or in the FOREACH statement of SPL):

- The FROM clause of the SELECT statement to access an element of the **collection** variable
- The INTO clause of the INSERT statement to add a new element to the **collection** variable
- The DELETE statement to remove an element from the **collection** variable
- The UPDATE statement to modify an existing element in the **collection** variable
- The DECLARE statement to declare a Select or Insert cursor to access multiple elements of the **collection** host variable
- The FETCH statement to retrieve a single element from a **collection** host variable that is associated with a Select cursor
- The PUT statement to retrieve a single element from a **collection** host variable that is associated with an Insert cursor
- The FOREACH statement to declare a cursor to access multiple elements of an SPL collection variable and to retrieve a single element from this **collection** variable

Using a Collection Variable to Manipulate Collection Elements

About this task

When you use data manipulation statements (SELECT, INSERT, UPDATE, or DELETE) of Informix® in conjunction with a **collection** variable, you can modify one or more elements in a collection.

To modify elements in a collection

1. Create a **collection** variable in your SPL routine or program.
   For information on how to declare a **collection** variable in , see the *HCL® Informix® Enterprise Replication Guide*. For information on how to define a **COLLECTION** variable in SPL, see DEFINE on page 920.

2. In , allocate memory for the collection; see ALLOCATE COLLECTION statement on page 18.
3. Optionally, use a SELECT statement to select a COLLECTION column into the **collection** variable.
   If the variable is an untyped COLLECTION variable, you must perform a SELECT from the COLLECTION column before you use the variable in the collection-derived table segment. The SELECT statement allows the database server to obtain the collection data type.
4. Use the appropriate data manipulation statement with the collection-derived table segment to add, delete, or update elements in the collection variable.
   To insert more than one element or to update or delete a *specific* element of a collection, you must use a cursor for the collection variable.
   - For more information on how to use an update cursor with ESQL/C, see DECLARE statement on page 433.
   - For more information on how to use an update cursor with SPL, see FOREACH on page 935.
5. After the collection variable contains the correct elements, use an INSERT or UPDATE statement on the table or view that holds the actual collection column to save the changes that the collection variable holds.
   - With UPDATE, specify the collection variable in the SET clause.
   - With INSERT, specify the collection variable in the VALUES clause.

Results

The collection variable stores the elements of the collection. It has no intrinsic connection, however, with a database column. Once the collection variable contains the correct elements, you must then save the variable into the actual collection column of the table with either an INSERT or an UPDATE statement.

Example of Deleting from a Collection in ESQL/C

Suppose that the **set_col** column of a row in the **table1** table is defined as a SET and for one row contains the values {`1,8,4,5,2`}. The following code fragment uses an update cursor and a DELETE statement with a WHERE CURRENT OF clause to delete the element whose value is `4`:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(smallint not null) a_set;
   int an_int;
EXEC SQL END DECLARE SECTION;
...
EXEC SQL allocate collection :a_set;
EXEC SQL select set_col into :a_set from table1 where int_col = 6;
EXEC SQL declare set_curs cursor for
   select * from table(:a_set) for update;

EXEC SQL open set_curs;
while (i<coll_size)
{
   EXEC SQL fetch set_curs into :an_int;
   if (an_int = 4)
   {
      EXEC SQL delete from table(:a_set) where current of set_curs;
      break;
   }
   i++;
}
```

```
EXEC SQL update table1 set set_col = :a_set
   where int_col = 6;
EXEC SQL deallocate collection :a_set;
EXEC SQL close set_curs;
EXEC SQL free set_curs;
```

After the DELETE statement executes, this collection variable contains the elements {1,8,5,2}. The UPDATE statement at the end of this code fragment saves the modified collection into the **set_col** column. Without this UPDATE statement, element 4 of the collection column is not deleted.

Example of Deleting from a Collection

Suppose that the **set_col** column of a row in the **table1** table is defined as a SET and one row contains the values {1,8,4,5,2}. The following SPL code fragment uses a FOREACH loop and a DELETE statement with a WHERE CURRENT OF clause to delete the element whose value is 4:

```
CREATE_PROCEDURE test6()

   DEFINE a SMALLINT;
   DEFINE b SET(SMALLINT NOT NULL);
   SELECT set_col INTO b FROM table1
      WHERE id = 6;
      -- Select the set in one row from the table
      -- into a collection variable
   FOREACH cursor1 FOR
      SELECT * INTO a FROM TABLE(b);
         -- Select each element one at a time from
         -- the collection derived table b into a
      IF a = 4 THEN
         DELETE FROM TABLE(b)
            WHERE CURRENT OF cursor1;
            -- Delete the element if it has the value 4
         EXIT FOREACH;
      END IF;
   END FOREACH;

   UPDATE table1 SET set_col = b
      WHERE id = 6;
      -- Update the base table with the new collection

END PROCEDURE;
```

This SPL routine declares two SET variables, **a** and **b**, each to hold a set of SMALLINT values. The first SELECT statement copies a SET column from one row of **table1** into variable **b**. The routine then declares a cursor called **cursor1** that copies one element at a time from **b** into SET variable **a**. When the cursor is positioned on the element whose value is 4, the DELETE statement removes that element from SET variable **b**. Finally, the UPDATE statement replaces the row of **table1** with the new collection that is stored in variable **b**.

For information on how to use collection variables in an SPL routine, see the *HCL® Informix® Guide to SQL: Tutorial*.

Example of Updating a Collection

Suppose that the **set_col** column of a table called **table1** is defined as a SET and that it contains the values {`1,8,4,5,2`}. The following program changes the element whose value is `4` to a value of `10`:

```
main
{
   EXEC SQL BEGIN DECLARE SECTION;
      int a;
      collection b;
   EXEC SQL END DECLARE SECTION;

   EXEC SQL allocate collection :b;
   EXEC SQL select set_col into :b from table1
      where int_col = 6;

   EXEC SQL declare set_curs cursor for
      select * from table(:b) for update;
   EXEC SQL open set_curs;
   while (SQLCODE != SQLNOTFOUND)
   {
      EXEC SQL fetch set_curs into :a;
      if (a = 4)
      {
         EXEC SQL update table(:b)(x)
            set x = 10 where current of set_curs;
         break;
      }
   }
   EXEC SQL update table1 set set_col = :b
      where int_col = 6;
   EXEC SQL deallocate collection :b;
   EXEC SQL close set_curs;
   EXEC SQL free set_curs;
}
```

After you execute this program, the **set_col** column in **table1** contains the values {`1,8,10,5,2`}.

This program defines two **collection** variables, **a** and **b**, and selects a SET from **table1** into **b**. The WHERE clause ensures that only one row is returned. Then the program defines a Collection cursor, which selects elements one at a time from **b** into **a**. When the program locates the element with the value `4`, the first UPDATE statement changes that element value to `10` and exits the loop.

In the first UPDATE statement, **x** is a derived-column name used to update the current element in the collection-derived table. The second UPDATE statement updates the base table **table1** with the new collection.

For information on how to use **collection** host variables in programs, see the discussion of complex data types in the *HCL® Informix® Enterprise Replication Guide*.

Example of Inserting a Value into a Multiset Collection

Suppose the host variable **a_multiset** has the following declaration:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection multiset(integer not null) a_multiset;
EXEC SQL END DECLARE SECTION;
```

The following INSERT statement adds a new MULTISET element of 142,323 to **a_multiset**:

```
EXEC SQL allocate collection :a_multiset;
EXEC SQL select multiset_col into :a_multiset from table1
   where id = 107;
EXEC SQL insert into table(:a_multiset) values (142323);
EXEC SQL update table1 set multiset_col = :a_multiset
   where id = 107;

EXEC SQL deallocate collection :a_multiset;
```

When you insert elements into a **client**-**collection** variable, you cannot specify a SELECT statement or an EXECUTE FUNCTION statement in the VALUES clause of the INSERT. When you insert elements into a **server**-**collection** variable, however, the SELECT and EXECUTE FUNCTION statements are valid in the VALUES clause. For more information on **client**- and **server**-**collection** variables, see the *HCL® Informix® Enterprise Replication Guide*.

Accessing a Nested Collection

If the element of the collection is itself a complex type (**collection** or **row** type), the collection is a *nested collection*. For example, suppose the **collection** variable, **a_set**, is a nested collection that is defined as follows:

```
EXEC SQL BEGIN DECLARE SECTION;
   client collection set(list(integer not null)) a_set;
   client collection list(integer not null) a_list;
   int an_int;
EXEC SQL END DECLARE SECTION;
```

To access the elements (or fields) of a nested collection, use a **collection** or **row** variable that matches the element type (**a_list** and **an_int** in the preceding code fragment) and a Select cursor.

Accessing a Row Variable

The TABLE keyword can make the **row** variable a collection-derived table. That is, a row appears as a table in an SQL statement. For a **row** variable, think of the collection-derived table as a table of one row, with each field of the **row** type being a column of the row. Use the TABLE keyword in place of the name of a table, synonym, or view in these SQL statements:

- The FROM clause of the SELECT statement to access a field of the **row** variable
- The UPDATE statement to modify an existing field in the **row** variable

The DELETE and INSERT statements do not support a **row** variable in the collection-derived-table segment.

For example, suppose an ESQL/C host variable **a_row** has the following declaration:

```
EXEC SQL BEGIN DECLARE SECTION;
   row(x int, y int, length float, width float) a_row;
EXEC SQL END DECLARE SECTION;
```

The following ESQL/C code fragment adds the fields in the **a_row** variable to the **row_col** column of the **tab_row** table:

```
EXEC SQL update table(:a_row)
   set x=0, y=0, length=10, width=20;
EXEC SQL update rectangles set rect = :a_row;
```

Database Name

Use the Database Name segment to specify the name of a database. Use this segment when you see a reference to a database name in a syntax diagram.

Syntax

(explicit id unique_262_Connect_42_dbn001) unique_262_Connect_42_dbn001

Database Name

" { { *dbname* [*@dbservername*] | *'//dbservername/dbname'* } | ?*db_var* } "

Element	Description	Restrictions	Syntax
dbname	Database name (with no pathname nor database server name)	Must be unique among the names of databases of the database server	Identifier on page 1231
dbservername	Database server on which the database *dbname* resides	Must exist. No blank space can separate `@` from *dbservername*.	Identifier on page 1231
db_var	Host variable whose value specifies a database environment	Variable must be a fixed-length character data type	Language specific

Usage

Database names are not case sensitive. You cannot use delimited identifiers for a database name.

The identifiers *dbname* and *dbservername* can each have a maximum of 128 bytes.

If the name of a database server is a delimited identifier or if it includes uppercase letters, that database server cannot participate in cross-server distributed DML operations. To avoid this restriction, use only undelimited names that include no uppercase letters when you declare the name or the alias of a database server.

In a nondefault locale, *dbname* can include alphabetic characters from the code set of the locale. In a locale that supports a multibyte code set, keep in mind that the maximum length of the database name refers to the number of bytes, not the number of characters. For more information on the aspects of naming databases, see the *HCL® Informix® GLS User's Guide*.

Using Keywords as Table Names

You can choose a database on another database server as your current database by specifying a database server name. The database server that *dbservername* specifies must match the name of a database server that is listed in your **sqlhosts** information.

Using the @ Symbol

The `@` symbol is a literal character. If you specify a database server name, blank spaces are not valid between the @ symbol and the database server name. Either put a blank space between *dbname* and the @ symbol, or omit the blank space.

The following examples show valid database specifications, qualified by the database server name:

```
empinfo@personnel
empinfo @personnel
```

In these examples, **empinfo** is the name of the database and **personnel** is the name of the database server.

Using a Path-Type Naming Notation

If you specify a pathname, do not put blank spaces between the quotation marks, slashes, and names. The following example specifies a valid UNIX™ pathname:

```
'//personnel/empinfo'
```

Here **empinfo** is the *dbname* and **personnel** is the name of the database server.

Using a Host Variable

You can use a host variable within an ESQL/C application to store a value that represents a database environment.

Database Object Name

Use the Database Object Name segment to specify the name of a database object, such as a column, table, view, or user-defined routine. Use this segment whenever you see a reference to a database object name.

Syntax

(explicit id unique_438_Connect_42_dobjn101) unique_438_Connect_42_dobjn101(explicit id unique_438_Connect_42_dobjn103)

unique_438_Connect_42_dobjn103

Database Object Name

" [[*database* [*@dbservername*] :] "

" [<Owner Name> [.] "

" *object* [. *object*] "

Element	Description	Restrictions	Syntax
database	Database where *object* resides	Must exist.	Database Name on page 1224
dbservername	Database server of *database*	Must exist. No space after `@`.	Identifier on page 1231
object	Name of a database object	See Usage on page 1226.	Identifier on page 1231

Usage

A database object name can include qualifiers and separator symbols to specify a database, a server, an owner, and (for some objects) another object of which the current database object is a component. For example, this expression specifies the **unit-price** column of the **stock** table, owned by user **informix**, in the **stores_demo** database of a database server called **butler**:

```
stores_demo@butler:informix.stock.unit_price
```

If you are creating or renaming a database object, the new name that you declare must be unique among objects of the same type in the database. Thus, the name of a new view must be unique among the names and synonyms of tables, views, and sequence objects that already exist in the same database. (But a view can have the same name as a view in a different database of the same server, or the same name as a trigger, for example, because these are different types of objects.)

In an ANSI-compliant database, the *owner*.*object* combination must be unique in the database for the type of object. A database object specification must include the owner name for a database object that you do not own. For example, if you specify a table that you do not own, you must also specify the owner of the table. The owner of all the system catalog tables is **informix**.

In Informix®, the uniqueness requirement does not apply to the name of a user defined routine (UDR). For more information, see Routine Overloading and Routine Signatures on page 1228.

Characters from the code set of your database locale are valid in database object names. For more information, see *HCL® Informix® GLS User's Guide*.

Specifying a Database Object in an External Database

Besides objects in the local database to which you are currently connected, you can also specify a database object in another database of the local database server, or in a database of a remote database server.

Specifying a Database Object in a Cross-Database Query

To specify an object in another database of the local database server, you must qualify the identifier of the object with the name of the database (and of the owner, if the external database is ANSI compliant), as in this example:

```
corp_db:hrdirector.executives
```

In this example, the name of the external database is **corp_db**. The name of the owner of the table is **hrdirector**. The name of the table is **executives**. Here the colon (**:**) separator is required after the *database* qualifier.

In Informix®, queries and other data manipulation language (DML) operations on other databases of the local database server can access most of the *built-in opaque data types*, as listed in Data Types in Cross-Database Transactions on page 700. DML operations can also access user-defined data types (UDTs) that can be cast to built-in types, as well as DISTINCT types that are based on built-in types, if each DISTINCT types and UDT is cast explicitly to a built-in type, and if all the DISTINCT types, UDTs, and casts are defined in all of the participating databases. The same data-type restrictions also apply to the arguments and to the returned values of a user-defined routine (UDR) that accesses other databases of the local Informix® instance, if the UDR is defined in all of the participating databases.

Specifying a Database Object in a Cross-Server Query

To specify an object in a database of a remote database server, you must use a *fully-qualified identifier* that specifies the database, the database server instance, and the owner (if the external database is ANSI compliant), in addition to the database object name.

For example, **hr_db@remoteoffice:hrmanager.employees** is a fully-qualified table name.

- Here the database name is **hr_db**,
- the **@** separator (ASCII 64), with no blank spaces, is required between the *database* and *database server* qualifiers,
- the database server name is **remoteoffice**,
- the **:** separator (ASCII 58), with no blank spaces, is required between the *database server* and *owner* qualifiers,
- the authorization identifier of the table *owner* is **hrmanager**,
- the **.** separator (ASCII 46), with no blank spaces, is required between the *owner* and *table* qualifiers,
- and the *table* name is **employees**.

Cross-server queries can access columns of built-in data types that are not opaque data types, but they cannot access UDTs nor complex data types. Among built-in opaque types, only BOOLEAN, BSON, JSON, VARCHAR, and DISTINCT types based on those base types, can be accessed or returned by DML statements or by UDRs in the databases of remote server instances. (For more information about the DISTINCT and built-in OPAQUE data types that Informix® supports in cross-server operations, see Data Types in Cross-Server Transactions on page 702.)

If a UDR exists on a remote database server instance, you must specify a fully-qualified identifier for the UDR. The following SQL statement invokes a routine in the same remote database as the previous example, where the authorization identifier of the owner of the routine is **johan**, and the identifier of the UDR is **suggestion_box**:

```
EXECUTE FUNCTION hr_db@remoteoffice:johan.suggestion_box(0):
```

You can refer to a remote database object only in the following SQL statements:

- CREATE DATABASE
- CREATE SYNONYM
- CREATE VIEW
- DATABASE
- DELETE
- EXECUTE FUNCTION
- EXECUTE PROCEDURE

- INFO
- INSERT
- LOAD
- LOCK TABLE
- MERGE
- SELECT
- UNLOAD
- UNLOCK TABLE
- UPDATE

In distributed MERGE operations, source tables can be in the database of a remote server instance, but the target table must be in a database of the local Informix instance.

For information on the support in these statements across databases of the local server, or across database servers, refer to the *HCL® Informix® Guide to SQL: Tutorial*.

If the name of a database server is a delimited identifier or if it includes uppercase letters, that database server cannot participate in distributed DML operations. To avoid this restriction, use only undelimited names that include no uppercase letters when you declare the name or the alias of a database server.

Restrictions on remote user-defined routines

The EXECUTE FUNCTION or EXECUTE PROCEDURE statement can invoke a routine in the database of a remote server instance in most contexts where a UDR is valid in the local database, but every participating database must have the same logging mode.

Cross-server operations require exactly one *coordinator* to connect to one or more database server instances as subordinate *participants* that process branches of the operation that reads or modifies objects in their databases.

Only the database server instance that initiated the cross-server operation as the coordinator can establish a connection with another database server. A distributed operation that invokes UDRs on one or more remote database servers can have a logical topology of connections resembling a star schema, but not a snowflake schema. In the one-to-many relationship between the coordinator and participants, only the coordinator can be the calling context for invoking remote UDRs.

If the EXECUTE FUNCTION or EXECUTE PROCEDURE statement calls a remote routine that in turn attempts to execute a remote routine on a third database server instance, the distributed operation fails with error -556. A UDR called from a participant server cannot invoke a UDR on a third server, and cannot reference any object on a third server.

Routine Overloading and Routine Signatures

Because of routine overloading, the name of a user-defined routine does not need to be unique to the database. You can define more than one UDR with the same name, provided that the *routine signature* for each UDR is different.

UDRs are uniquely identified by their signatures. The signature of a UDR includes the following items of information:

- The type of routine (function or procedure)
- The identifier of the routine
- The cardinality, data type, and order of the parameters
- In an ANSI-compliant database, the owner name

For any given UDR, at least one item in the routine signature must be unique among all the UDRs registered in the database.

In a database that is not ANSI-compliant, two routines that have different owners cannot have the same signature, except in the special cases of the **sysdbopen( )** and **sysdbclose( )** routines. For information about the effects of these session configuration routines when their owners connect to or disconnect from a database where these routines are defined, see IFX_REPLACE_MODULE Function.

Specifying an Existing UDR

To reference an existing UDR by a name that does not uniquely identify the UDR, you must also specify the parameter data types after the UDR name, in the same order that they were declared when the UDR was created. Informix® then uses routine resolution rules to identify the instance of the UDR to alter, drop, or execute. As an alternative, you can specify its *specific name*, if one was declared when the UDR was created. Specific names are described in the section Specific Name on page 1288. For more details of routine resolution, see Comparing Arguments to the Parameter List on page 1210, and *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

Owners of Objects Created by UDRs

When a DDL statement within an owner-privileged UDR creates a new database object, the owner of the routine (rather than the user who executes it, if that user is not the owner of the routine) becomes the owner of the new database object. For a DBA-privileged UDR, however, the user who executes the routine (and who must hold DBA privilege) becomes the owner of any objects that the UDR creates.

External Routine Reference

Include the External Routine Reference clause when you write an external routine. This option is not available for SPL routines.

Syntax

(explicit id unique_311_Connect_42_exrr002) unique_311_Connect_42_exrr002(explicit id unique_311_Connect_42_exrr003)

unique_311_Connect_42_exrr003(explicit id unique_311_Connect_42_exrr004) unique_311_Connect_42_exrr004

External Routine Reference

" EXTERNAL NAME "

" <Shared-Object Filename>[] "

" LANGUAGE "

" { C | JAVA } "

" [PARAMETER STYLE INFORMIX] "

" [[] { VARIANT | NOT VARIANT }] "

Usage

If the IFX_EXTEND_ROLE configuration parameter is set to `ON` or to `1`, authorization to use this segment is available only to the Database Server Administrator (DBSA), and to users whom the DBSA has granted the EXTEND role. By default, the DBSA is user **informix**. In addition, you cannot create an external routine unless you hold the Resource or DBA privilege on the database, and also hold the Usage privilege on the external programming language in which the routine is written. For the syntax of the GRANT USAGE ON LANGUAGE C and GRANT USAGE ON LANGUAGE JAVA statements of SQL, see Language-Level Privileges on page 549.

This segment specifies the following information about an external routine:

- Pathname to the executable object code, stored in a shared-object file

  For C routines, this file is either a DLL or a shared library, depending on your operating system.

  For Java™ routines, this file is a jar file. Before you can create a UDR written in the Java™ language, you must assign a jar identifier to the external jar file with the **sqlj.install_jar** procedure. For more information, see sqlj.install_jar.
- The name of the programming language in which the UDR is written
- The parameter style of the UDR

  By default, the parameter style is INFORMIX. (This implies that if you specify OUT or INOUT parameters, the OUT or INOUT values are passed by reference.)

- The VARIANT or NOT VARIANT option. If you specify neither, the default is VARIANT. If the routine includes any statement of SQL it is a VARIANT routine. If a DDL statement that includes the External Routine Reference clause also includes the Routine Modifier clause, do not classify the same UDR as VARIANT in one of these clauses and as NOT VARIANT in the other.

Example

The following example includes an external routine reference for a Java™ language UDR. You must first register **demo_jar**using the procedure **install_jar**(<*absolute path*><*jar file name*>,<*internal registered name*>).

```
CREATE FUNCTION delete_order(int) RETURNING int
   EXTERNAL NAME 'informix.demo_jar:delete_order.delete_order()'
   LANGUAGE JAVA;
```

VARIANT or NOT VARIANT Option

A function is *variant* if it can return different results when it is invoked with the same arguments or if it modifies the state of a database or of a variable. For example, a function that returns the current date or time is a variant function.

By default, user-defined functions are variant. If you specify NOT VARIANT when you create or modify a function, it cannot contain any SQL statements.

If the function is nonvariant, the database server might cache the return variant functions. For more information on functional indexes, see CREATE INDEX statement on page 237.

To register a nonvariant function, add the NOT VARIANT option in this clause or in the Routine Modifier clause that is discussed in Routine modifier on page 1275. If you specify the modifier in both contexts, however, you must use the same modifier (either VARIANT or NOT VARIANT) in both clauses.

Example of a C User-Defined Function

The next example registers an external function named **equal( )** that takes two **point** data type values as arguments. In this example, **point** is an opaque data type that specifies the **x** and **y** coordinates of a two-dimensional point.

```
CREATE FUNCTION equal( a point, b point ) RETURNING BOOLEAN;
   EXTERNAL NAME "/usr/lib/point/lib/libbtype1.so(point1_equal)"
   LANGUAGE C
END FUNCTION;
```

The function returns a single value of type BOOLEAN. The external name specifies the path to the C shared-object file where the object code of the function is stored. The external name indicates that the library contains another function, **point1_equal( )**, which is invoked while **equal( )** executes.

Identifier

An *identifier* specifies the unqualified name of a database object, such as an access method, aggregate, alias, blobspace, cast, column, constraint, correlation, data type, index, operator class, partition, procedure, table, trigger, sequence, synonym, or view. Use the Identifier segment whenever you see a reference to an identifier in a syntax diagram.

Syntax

(explicit id unique_8_Connect_42_id003) unique_8_Connect_42_id003

Identifier

" { { *letter* | *underscore* } [{ | *letter* | *digit* | *underscore* | *dollar_sign* }] | <Delimited Identifier> }

Element	Description	Restrictions	Syntax
digit	Integer in range `0` to `9`	Cannot be the first character	Literal Number on page 1198
dollar_sign	Dollar ($) symbol	Cannot be the first character	Literal symbol entered from the keyboard.
letter	Upper- or lowercase letter of the alphabet	In the default locale, must be an ASCII character in the range A to Z or a to z	Literal symbol entered from the keyboard.
underscore	Underscore (_) character	Cannot substitute a space, hyphen, or other non-alphanumeric character	Literal symbol entered from the keyboard.

Usage

This is a logical subset of Database Object Name on page 1225, a segment that can specify the *owner*, *database*, and *database server* of external objects.

To include other non-alphanumeric symbols, such as a blank space (ASCII 32), in an identifier, you must use a delimited identifier. It is recommended that you do not use the dollar sign ($) in identifiers, because this symbol is a special character whose inclusion in an identifier might cause conflicts with other syntax elements. For more information, see Delimited Identifiers on page 1233.

An identifier must have a length of at least 1 byte, but no more than 128 bytes. For example, **employee_information** is valid as a table name. If you are using a multibyte code set, keep in mind that the maximum length of an identifier refers to the number of bytes, not to the number of logical characters.

For letter characters in nondefault locales, see Support for Non-ASCII Characters in Identifiers on page 1233. For further information on the aspects of identifiers, see Chapter 3 of the *HCL® Informix® GLS User's Guide*.

When you use ESQL/C with Informix® , the database server checks the internal version number of the client application and the setting of the **IFX_LONGID** environment variable to determine whether a client application supports long identifiers (up to 128 bytes in length). For more information, see the *HCL® Informix® Guide to SQL: Reference*.

When the database server uses long identifiers, you might encounter error messages, warning messages, or other messages that truncate trailing characters in SQL identifiers or elsewhere in the message text. Truncation can usually be avoided, however, if identifiers have 18 or fewer bytes. Your code might be difficult to read or to maintain if identifiers of different SQL objects are identical in their first 18 characters.

Use of Uppercase Characters

You can specify the name of a database object with uppercase characters, but the database server shifts these to lowercase characters unless the **DELIMIDENT** environment variable is set and the identifier of the database object is enclosed between double (") quotation marks. In this case, the database server treats the name of the database object as a delimited identifier and preserves the uppercase characters in the name, as described in Delimited Identifiers on page 1233.

If the name of a database server includes uppercase letters, that database server cannot participate in distributed DML operations. To avoid this restriction, use only undelimited names that include no uppercase letters when you declare the name or the alias of a database server.

Use of Keywords as Identifiers

Although you can use almost any word as an identifier, syntactic ambiguities can result from using keywords as identifiers in SQL statements. The statement might fail or might not produce the expected results. For a discussion of the syntactic ambiguities that can result from using keywords as identifiers and an explanation of workarounds for these problems, see Potential Ambiguities and Syntax Errors on page 1236.

Delimited identifiers provide the easiest and safest way to use a keyword as an identifier without syntactic ambiguities. No workarounds are necessary for a keyword as a delimited identifier. For the syntax and usage of delimited identifiers, see Delimited Identifiers on page 1233. Delimited identifiers require, however, that your code always use single (') quotation marks, rather than double (") quotation marks, to delimit character-string literals.

For the keywords of the implementation of SQL in Informix®, see Keywords of SQL for HCL Informix on page 1295.

Tip: If an error message seems unrelated to the statement that caused the error, check to see if the statement uses a keyword as an undelimited identifier.

Support for Non-ASCII Characters in Identifiers

In a nondefault locale, you can use any alphabetic character that your locale recognizes as a *letter* in an SQL identifier. This feature enables you to use non-ASCII characters in the names of some database objects. For objects that support non-ASCII characters, see the *HCL® Informix® GLS User's Guide*.

Delimited Identifiers

By default, the character set of a valid SQL identifier is restricted to letters, digits, underscore, and dollar-sign symbols. If you set the **DELIMIDENT** environment variable, however, SQL identifiers can also include additional characters from the code set implied by the setting of the **DB_LOCALE** environment variable.

Delimited Identifier

" " "

" { | *letter* | *digit* | *underscore* | *other_character* } "

" " "

Element	Description	Restrictions	Syntax
digit	Integer in the range `0` to `9`	Cannot be the first character	Literal Number on page 1198
letter	Letter that forms part of the delimited identifier	Letters in delimited identifiers are case-sensitive	Literal value entered from the keyboard.
other_character	Nonalphanumeric character, such as #, $, or blank space	Must be an element in the code set of the database locale	Literal value entered from the keyboard.
underscore	Underscore (_) symbol in the delimited identifier	Cannot include more than 128	Literal value entered from the keyboard.

If the database supports delimited identifiers, any double quotation marks (`"`) enclose an SQL identifier in your code, and only single (`'`) quotation marks, rather than double (`"`) quotation marks, delimit character-string literals.

Delimited identifiers enable you to declare names that are otherwise identical to SQL keywords, such as TABLE, WHERE, DECLARE, and so on. The only type of object for which you cannot specify a delimited identifier is a database name.

Letters in delimited identifiers are case sensitive. If you are using the default locale, *letter* must be an upper- or lowercase character in the range a to z or A to Z (in the ASCII code set). If you are using a nondefault locale, *letter* must be an alphabetic character that the locale supports. For more information, see Support for Non-ASCII Characters in Delimited Identifiers (GLS) on page 1235.

Delimited identifiers are compliant with the ANSI/ISO standard for SQL.

When you create a database object, avoid including leading blank spaces or other white-space characters between the first delimiting quotation mark and the first nonblank character of the delimited identifier. (Otherwise, you might not be able to reference the object in some contexts.)

If the name of a database server is a delimited identifier or if it includes uppercase letters, that database server cannot participate in distributed DML operations. To avoid this restriction, use only undelimited names that include no uppercase letters when you declare the name or the alias of a database server.

Support for Nonalphanumeric Characters

By default, ASCII letters, digits, and the underscore (ASCII 95) character are supported in SQL identifiers and in storage object identifiers for all locales. To include additional characters from the codeset implied by the **DB_LOCALE** setting in the names of database objects, you must use delimited identifiers.

You cannot , however, use delimited identifiers, however, to specify characters that are not letters, digits, or the underscore (_) character when you declare or reference the names of storage objects, such as dbspaces, partitions, blobspaces, or sbspaces.

Support for Non-ASCII Characters in Delimited Identifiers (GLS)

When you are using a nondefault locale whose code set supports non-ASCII characters, you can specify those non-ASCII characters in most delimited identifiers. The rule is that if you can specify non-ASCII characters in the undelimited form of the identifier, you can also specify non-ASCII characters in the delimited form of the same identifier. For a list of identifiers that support non-ASCII characters and for information on non-ASCII characters in delimited identifiers, see the *HCL® Informix® GLS User's Guide*.

Enabling Delimited Identifiers

About this task

To use delimited identifiers, you must set the **DELIMIDENT** environment variable. While **DELIMIDENT**is set, strings enclosed in double quotation marks (") are treated as identifiers of database objects, and strings enclosed in single quotation marks (') are treated as literal strings. If the **DELIMIDENT** environment variable is not set, however, strings enclosed in double quotation marks are also treated as literal strings.

If **DELIMIDENT** is set, the SELECT statement in the following example must be in single quotation marks in order to be treated as a quoted string:

```
PREPARE ... FROM 'SELECT * FROM customer';
```

If a delimited identifier is used in the SELECT statement that defines a view, then the **DELIMIDENT** environment variable must be set in order for the view to be accessed, even if the view name itself contains no special characters.

On UNIX™ and Linux™ systems, you can set **DELIMIDENT** by the procedures for setting environment variables that are described in *HCL® Informix® Guide to SQL: Reference*.

On Windows™ systems, you can set **DELIMIDENT** in various ways, which generally have the following descending order of precedence:

1. The setting of **DELIMIDENT** in the connection string when connecting
2. The setting of the SQL_INFX_ATTR_DELIMIDENT connection attribute before connecting
3. The setting of **DELIMIDENT** in **setnet32** with the **Use my settings** box selected
4. The setting of **DELIMIDENT** in **setnet32** with the **Use my settings** box cleared
5. The setting of **DELIMIDENT** on the command line before running the application
6. The setting of **DELIMIDENT** in Windows™ as a user variable

7. The setting of **DELIMIDENT** in Windows™ as a system variable
8. The default value (of no support for delimited identifiers).

This general order of precedence for Windows™ clients is sensitive, however, to the API through which you connect to the database, which can also affect the meaning of the setting and the default value. Refer to the documentation of your specific API for more information about the **DELIMIDENT** setting in Windows™.

Using Double Quotation Marks in a Delimited Identifier

To include a double quotation mark (") character within a delimited identifier, you must precede the double quotation mark (") with another double quotation mark ("). The following statement fragment specifies **My "Good" Data** as a table name:

```
CREATE TABLE "My ""Good"" Data" (...);
```

Potential Ambiguities and Syntax Errors

IBM® does not recommend using any keyword of SQL as an identifier, because to do so tends to make your code more difficult to read and to maintain. If you ignore this potential problem for human readers, however, you can use almost any keyword as an SQL identifier, but various syntactic ambiguities can occur. An ambiguous statement might not produce the desired results. The following sections identify some potential ambiguities and workarounds when keywords are declared as identifiers, or when different database objects have the same identifier.

Using the Names of Built-In Functions as Column Names

The following two examples show a workaround for using a built-in function as a column name in a SELECT statement. This workaround applies to the built-in aggregate functions (AVG, COUNT, MAX, MIN, SUM) as well as the function expressions (algebraic, exponential and logarithmic, time, HEX, length, DBINFO, trigonometric, and TRIM functions).

Using **avg** as a column name causes the next example to fail because the database server interprets **avg** as an aggregate function rather than as a column name:

```
SELECT avg FROM mytab; -- fails
```

If the **DELIMIDENT** environment variable is set, you could use **avg** as a column name as the following example shows:

```
SELECT "avg" from mytab; -- successful
```

The workaround in the following example removes ambiguity by including a table name with the column name:

```
SELECT mytab.avg FROM mytab;
```

If you use the keyword TODAY, CURRENT, SYSDATE, or USER as a column name, ambiguity can occur, as the following example shows:

```
CREATE TABLE mytab (user char(10),
   CURRENT DATETIME HOUR TO SECOND,TODAY DATE);

INSERT INTO mytab VALUES('josh','11:30:30','1/22/2008');

SELECT user,current,today FROM mytab;
```

The database server interprets **user**, **current**, and **today** in the SELECT statement as the built-in functions USER, CURRENT, and TODAY. Thus, instead of returning `josh`, `11:30:30`,`1/22/2008`, the SELECT statement returns the current user name, the current time, and the current date. The SYSDATE keyword has a similar effect in databases of Informix®.

If you want to select the actual columns of the table, you must write the SELECT statement in one of the following ways:

```
SELECT mytab.user,  mytab.current,  mytab.today FROM mytab;

EXEC SQL select * from mytab;
```

Using Keywords as Column Names

Specific workarounds exist for using a keyword as a column name in a SELECT statement or other SQL statement. In some cases, more than one suitable workaround might be available.

Using ALL, DISTINCT, or UNIQUE as a Column Name

If you want to use the ALL, DISTINCT, or UNIQUE keywords as column names in a SELECT statement, you can take advantage of a workaround.

First, consider what happens when you try to use one of these keywords without a workaround. In the following example, using **all** as a column name causes the SELECT statement to fail because the database server interprets **all** as a keyword rather than as a column name:

```
SELECT all FROM mytab -- fails;
```

You must use a workaround to make this SELECT statement execute successfully. If the **DELIMIDENT** environment variable is set, you can use **all** as a column name by enclosing **all** in double quotation marks. In the following example, the SELECT statement executes successfully because the database server interprets **all** as a column name:

```
SELECT "all" from mytab; -- successful
```

The workaround in the following example uses the keyword ALL with the column name **all**:

```
SELECT ALL all FROM mytab;
```

The examples that follow show workarounds for using the keywords UNIQUE or DISTINCT as a column name in a CREATE TABLE statement.

The next example fails to declare a column named **unique** because the database server interprets **unique** as a keyword rather than as a column name:

```
CREATE TABLE mytab (unique INTEGER); -- fails
```

The following workaround uses two SQL statements. The first statement creates the column **mycol**; the second statement renames the column **mycol** to **unique**:

```
CREATE TABLE mytab (mycol INTEGER);

RENAME COLUMN mytab.mycol TO unique;
```

The workaround in the following example also uses two SQL statements. The first statement creates the column **mycol**; the second alters the table, adds the column **unique**, and drops the column **mycol**:

```
CREATE TABLE mytab (mycol INTEGER);

ALTER TABLE mytab
   ADD (unique INTEGER),
   DROP (mycol);
```

Using INTERVAL or DATETIME as a Column Name

The examples in this section show workarounds for using the keyword INTERVAL (or DATETIME) as a column name in a SELECT statement.

Using **interval** as a column name causes the following example to fail because the database server interprets **interval** as a keyword and expects it to be followed by an INTERVAL qualifier:

```
SELECT interval FROM mytab; -- fails
```

If the **DELIMIDENT** environment variable is set, you could use **interval** as a column name, as the following example shows:

```
SELECT "interval" from mytab; -- successful
```

The workaround in the following example removes ambiguity by specifying a table name with the column name:

```
SELECT mytab.interval FROM mytab;
```

The workaround in the following example includes an owner name with the table name:

```
SELECT josh.mytab.interval FROM josh.mytab;
```

Using rowid as a Column Name

Every nonfragmented table has a virtual column named **rowid**. To avoid ambiguity, you cannot use **rowid** as a column name. Performing the following actions causes an error:

- Creating a table or view with a column named **rowid**
- Altering a table by adding a column named **rowid**
- Renaming a column to **rowid**

You can, however, use the term **rowid** as a table name.

```
CREATE TABLE rowid (column INTEGER, date DATE, char CHAR(20));
```

Important: It is recommended that you use primary keys as an access method, rather than exploiting the **rowid** column.

Using keywords as table names

The database server issues an error in contexts where the unqualified identifier of a table object is also a valid keyword of SQL. You can disambiguate the table name by qualifying it with the authorization identifier of the owner of the table.

Examples that follow illustrate owner-name qualifiers as workarounds when the keyword `STATISTICS`, `OUTER`, or `FROM` has been declared as a table name or synonym. (These examples also apply if any of those keywords is the identifier of a view.)

Using **statistics** as a table identifier causes the following UPDATE statement example to fail. An exception occurs because the database server interprets **statistics** as a keyword in a syntactically incorrect UPDATE STATISTICS statement, rather than as the target table name in an UPDATE statement:

```
UPDATE statistics SET mycol = 10;  -- fails
```

The workaround in the following example qualifies the table name with the owner name, to avoid that ambiguity:

```
UPDATE josh.statistics SET mycol = 10;
```

Using **outer** as a table name causes the following example to fail, because the database server interprets **outer** as a keyword for performing a syntactically incorrect outer join:

```
SELECT mycol FROM outer; -- fails
```

The following successful example uses owner naming to avoid ambiguity:

```
SELECT mycol FROM josh.outer;
```

The following DELETE statement, whose target table was created with **from** as its identifier, returns a syntax error:

```
DELETE from;    -- fails
```

Because `FROM` is also an optional keyword that immediately precedes the name of the table whose records are to be destroyed, the database server expects a table name after `FROM`. Finding none, it issues an exception.

The following example, in contrast, deletes all the rows from what it correctly recognizes as the **from** table, because that table name is qualified by the name of its owner:

```
DELETE zelaine.from;
```

If the **DELIMIDENT** environment variable is set on the database server, an alternative workaround is to use double quotation marks (`"`) as delimiters.

```
DELETE "from";
```

This avoids the exception by indicating that **from** is an SQL identifier, rather than a string literal or an SQL keyword.

Despite the availability of these workarounds, your code will be easier for humans to read and to maintain if you avoid declaring SQL keywords as the identifiers of tables, views, or other database objects.

Workarounds that Use the Keyword AS

In some cases, although a statement is not ambiguous and the syntax is correct, the database server returns a syntax error. The preceding pages show existing syntactic workarounds for several situations. You can use the AS keyword to provide a workaround for the exceptions.

You can use the AS keyword in front of column labels or table aliases.

The following example uses the AS keyword with a column label:

```
SELECT column_name AS display_label FROM table_name;
```

The following example uses the AS keyword with a table alias:

```
SELECT select_list FROM table_name AS table_alias;
```

Using AS with Column Labels

The examples in this section show workarounds that use the AS keyword with a column label. The first two examples show how you can use the keyword UNITS (or YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, or FRACTION) as a column label.

Using **units** as a column label causes the next example to fail because the database server interprets it as part of an INTERVAL expression in which the **mycol** column is the operand of the UNITS operator:

```
SELECT mycol units FROM mytab;
```

The workaround in the following example includes the AS keyword:

```
SELECT mycol AS units FROM mytab;
```

The following example uses the AS or FROM keyword as a column label.

Using **as** as a column label causes the following example to fail because the database server interprets **as** as identifying **from** as a column label and thus finds no required FROM clause:

```
SELECT mycol as from mytab; -- fails
```

The following successful example repeats the AS keyword:

```
SELECT mycol AS as from mytab;
```

Using **from** as a column label causes the following example to fail because the database server expects a table name to follow the first **from**:

```
SELECT mycol from FROM mytab; -- fails
```

This example uses the AS keyword to identify the first **from** as a column label:

```
SELECT mycol AS from FROM mytab;
```

Using AS with Table Aliases

Examples in this section show workarounds that use the AS keyword with a table alias. The first pair shows how to use the ORDER, FOR, GROUP, HAVING, INTO, UNION, WITH, CREATE, GRANT, or WHERE keyword as a table alias.

Using **order** as a table alias causes the following example to fail because the database server interprets **order** as part of an ORDER BY clause:

```
SELECT * FROM mytab order; -- fails
```

The workaround in the following example uses the keyword AS to identify **order** as a table alias:

```
SELECT * FROM mytab AS order;
```

The next two examples show how to use the keyword WITH as a table alias.

Using **with** as a table alias causes the next example to fail because the database server interprets **with** as part of the WITH CHECK OPTION syntax:

```
EXEC SQL select * from mytab with; -- fails
```

The workaround in the following example uses the keyword AS to identify **with** as a table alias:

```
EXEC SQL select * from mytab as with; -- succeeds
```

The next two examples use the keyword CREATE as a table alias. Using **create** as a table alias causes the next example to fail because the database server interprets the keyword as part of the syntax to create a new database object, such as a table, synonym, or view:

```
EXEC SQL select * from mytab create; -- fails
```

```
EXEC SQL select * from mytab as create; -- succeeds
```

The workaround uses the keyword AS to identify **create** as a table alias. (Using grant as an alias would similarly fail, but is valid after the AS keyword.)

Fetching Cursors that have Keywords as Names

In a few situations, no workaround exists for the syntactic ambiguity that occurs when a keyword is used as an identifier in an SQL program.

In the following example, the FETCH statement specifies a cursor named **next**. The FETCH statement generates a syntax error because the preprocessor interprets **next** as a keyword, signifying the next row in the active set and expects a cursor name to follow **next**. This occurs whenever the keyword NEXT, PREVIOUS, PRIOR, FIRST, LAST, CURRENT, RELATIVE, or ABSOLUTE is used as a cursor name:

```
/* This code fragment fails */
EXEC SQL declare next cursor for
    select customer_num, lname from customer;
EXEC SQL open next;
EXEC SQL fetch next into :cnum, :lname;
```

Declaring keywords as names for variables in UDRs

If you use some keywords as identifiers for variables in a user-defined routine (UDR), you can create ambiguous syntax:

The following constant expressions and other SQL keywords might produce unexpected results as variable names.

- CURRENT
- CURRENT_ROLE
- CURRENT_USER
- DATETIME
- DEFAULT_ROLE
- GLOBAL
- INTERVAL
- NULL

- OFF
- OUT
- PROCEDURE
- SELECT
- SYSDATE
- TODAY
- UNITS
- USER

Using CURRENT, DATETIME, INTERVAL, and NULL in INSERT

A UDR cannot insert a variable that was declared using the CURRENT, DATETIME, INTERVAL, or NULL keywords as the name. For example, if you declare a variable called **null**, when you try to insert the value **null** into a column, you receive a syntax error, as the following example shows:

```
CREATE PROCEDURE problem()
. . .
DEFINE null INT;
LET null = 3;
INSERT INTO tab VALUES (null); -- error, inserts NULL, not 3
```

Using NULL and SELECT in a Condition

If you declare a variable with the name **null** or **select**, including it in a condition that uses the IN keyword is ambiguous. The following example shows three conditions that cause problems: in an IF statement, in a WHERE clause of a SELECT statement, and in a WHILE condition:

```
CREATE PROCEDURE problem()
. . .
DEFINE x,y,select, null, INT;
DEFINE pfname CHAR[15];
LET x = 3; LET select = 300;
LET null = 1;
IF x IN (select, 10, 12) THEN LET y = 1; -- problem if

IF x IN (1, 2, 4) THEN
SELECT customer_num, fname INTO y, pfname FROM customer
   WHERE customer IN (select , 301 , 302, 303); -- problem in

WHILE x IN (null, 2)     -- problem while
. . .
END WHILE;
```

You can use the variable **select** in an IN list if you ensure it is not the first element in the list. The workaround in the following example corrects the IF statement that the preceding example shows:

```
IF x IN (10, select, 12) THEN LET y = 1; -- problem if
```

No workaround exists to using **null** as a variable name and attempting to use that variable in an IN condition.

Declaring Keywords or Routine Names as SPL Variables

If you declare a variable with the same name as a keyword or as the name of a routine, ambiguities can occur. Informix® uses the following rules for resolving name conflicts among SPL variables, UDR names, and built-in SQL function names.

- Variable names that are declared in DEFINE statements take the highest precedence.
- User-defined routines defined in CREATE PROCEDURE or CREATE FUNCTION statements take precedence over built-in SQL functions.
- Procedures declared with the PROCEDURE keyword in the DEFINE statement take precedence over built-in SQL functions.
- Built-in SQL functions take precedence over SPL procedures that exist in the database but that are not explicitly identified as procedures in the DEFINE statement.

Do not use the name of a built-in SQL function as an SPL variable if you might need to invoke the SQL function. For example, do not declare a variable with the name **count** or **max**, if you might also need to call those aggregate functions.

Variables that Conflict with Column Names

If you use the same identifier for an SPL variable and a column name, then within the scope of reference of the variable, the database server interprets any instance of the unqualified identifier as a variable. To use the identifier to specify a column name, use *table.column* notation to qualify the column name with the table name. In the following example, the procedure variable **lname** is the same as the column name. In the following SELECT statement, **customer.lname** is a column in the database and **lname** is an SPL variable:

```
CREATE PROCEDURE table_test()
DEFINE lname CHAR(15);
LET lname = "Miller";
SELECT customer.lname FROM customer INTO lname
   WHERE customer_num = 502;
```

This example is valid, but relying on the rules of precedence of Informix® to resolve name conflicts between SPL variables and column names might make your code difficult for human readers to interpret and to maintain. An alternative to reusing the same identifier as a variable and as a column name is for the DEFINE statement to declare some prefix to the identifier, such as **v_lname** in this example, to indicate that this variable stores the value of the column **lname**.

Using ON, OFF, or PROCEDURE with TRACE

If you define an SPL variable called **on**, **off**, or **procedure**, and you attempt to use it in a TRACE statement, the value of the variable is not traced. Instead, the TRACE ON, TRACE OFF, or TRACE PROCEDURE statements execute. You can trace the value of the variable by specifying the variable in a more complex expression.

The following example shows both the ambiguous syntax and workarounds that use arithmetic or string expressions that evaluate to the variable:

```
DEFINE on, off, procedure INT;

TRACE on;                    --ambiguous
TRACE 0+ on;  --ok
```

```
TRACE off;                    --ambiguous
TRACE ''||off;--ok

TRACE procedure;  --ambiguous
TRACE 0+procedure;--ok
```

Using GLOBAL as the Name of a Variable

If you attempt to define a variable with the name **global**, the define operation fails. The syntax that the following example shows conflicts with the syntax for defining global variables:

```
DEFINE global INT; -- fails;
```

If the **DELIMIDENT** environment variable is set, you could use **global** as a variable name, as the following example shows:

```
DEFINE "global" INT; -- successful
```

Important: Although workarounds that the preceding sections show can avoid compilation or runtime syntax conflicts from keywords used as identifiers, keep in mind that such identifiers tend to make code more difficult to understand and to maintain.

Using EXECUTE, SELECT, or WITH as Cursor Names

Do not use an EXECUTE, SELECT, or WITH keyword as the name of a cursor. If you try to use one of these keywords as the name of a cursor in a FOREACH statement, the cursor name is interpreted as a keyword in the FOREACH statement. No workaround exists.

The following example does not work:

```
DEFINE execute INT;
FOREACH execute FOR SELECT col1  -- error, looks to parser like
    INTO var1 FROM tab1;         -- 'FOREACH EXECUTE PROCEDURE'
```

SELECT Statements in WHILE and FOR Statements

If you use a SELECT statement in a WHILE or FOR loop, and if you need to enclose it in parentheses, enclose the entire SELECT statement in a BEGIN…END statement block. The SELECT statement in the first WHILE statement in the following example is interpreted as a call to the procedure **var1**; the second WHILE statement is interpreted correctly:

```
DEFINE var1, var2 INT;
WHILE var2 = var1
   SELECT col1 INTO var3 FROM TAB -- error, interpreted as call var1()
   UNION
   SELECT co2 FROM tab2;
END WHILE;

WHILE var2 = var1
   BEGIN
      SELECT col1 INTO var3 FROM TAB -- ok syntax
      UNION
      SELECT co2 FROM tab2;
```

```
   END
END WHILE;
```

SET keyword in the ON EXCEPTION statement of SPL

If you include an SQL statement that begins with the keyword `SET` in the ON EXCEPTION statement, you must enclose it in a BEGIN ... END statement block.

The following list shows some of the SQL statements that begin with the keyword SET:

- SET AUTOFREE
- SET CONNECTION
- SET CONSTRAINTS
- SET DATASKIP
- SET DEBUG FILE
- SET DEFERRED_PREPARE
- SET DESCRIPTOR
- SET ENCRYPTION
- SET ENVIRONMENT
- SET EXPLAIN
- SET INDEXES
- SET ISOLATION
- SET LOCK MODE
- SET LOG
- SET OPTIMIZATION
- SET PDQPRIORITY
- SET ROLE
- SET STATEMENT CACHE
- SET TABLE
- SET TRANSACTION
- SET TRIGGERS

The following examples show the incorrect and correct use of a SET LOCK MODE statement inside an ON EXCEPTION statement.

The following ON EXCEPTION statement returns an error because the SET LOCK MODE statement is not enclosed in a BEGIN ... END statement block:

```
ON EXCEPTION IN (-107)
   SET LOCK MODE TO WAIT; -- error, value expected, not 'lock'
END EXCEPTION;
```

The following ON EXCEPTION statement executes successfully because the SET LOCK MODE statement is enclosed in a BEGIN ... END statement block:

```
ON EXCEPTION IN (-107)
   BEGIN
```

```
   SET LOCK MODE TO WAIT; -- ok
   END
END EXCEPTION;
```

Jar Name

Use the Jar Name segment to specify the name of a jar ID. Use this segment whenever you see a reference to Jar Name in a syntax diagram.

Syntax

Jar Name

" [[*database* .] *package* .] "

" *jar_id* "

Element	Description	Restrictions	Syntax
database	Database in which to install or access *jar_id*. Default is the current database.	Fully qualified *database*.*package*.*jar_id* identifier must not exceed 255 bytes	Database Name on page 1224
jar_id	The **.jar** file that contains the Java™ class to be accessed	File must exist in *database*.*package*	Identifier on page 1231
package	Name of the package	Package must exist in *database*	Identifier on page 1231

If a jar name is specified as a character string argument to the **sqlj.install_jar**, **sqlj.replace_jar**, or **sqlj.remove_jar** procedures, then any identifiers in the jar name that are delimited identifiers will include the surrounding double quotation mark characters. For descriptions of the these procedures, see related concept SQLJ Driver Built-In Procedures.

Before you can access a *jar_id* in any way (including its use in a CREATE FUNCTION or CREATE PROCEDURE statement), it must be defined in the current database with the **install_jar( )** procedure. For more information, see EXECUTE PROCEDURE statement on page 510.

Optimizer Directives

The Optimizer Directives segment specifies keywords that you can use to partially or fully specify the query plan of the optimizer. Use this segment whenever you see a reference to Optimizer Directives in a syntax diagram.

Syntax

(explicit id unique_14_Connect_42_opdi004) unique_14_Connect_42_opdi004(explicit id unique_14_Connect_42_opdi005)

unique_14_Connect_42_opdi005(explicit id unique_14_Connect_42_opdi006) unique_14_Connect_42_opdi006(explicit id unique_14_Connect_42_opdi007)

unique_14_Connect_42_opdi007(explicit id unique_14_Connect_42_opdi008) unique_14_Connect_42_opdi008(explicit id unique_14_Connect_42_opdi010)

unique_14_Connect_42_opdi010(explicit id unique_14_Connect_42_opdi011) unique_14_Connect_42_opdi011

Optimizer Directives

"{ --+ |{+ |/*+ }"

"{| <Access-Method Directives> | <Join-Order Directive> | <Join-Method Directives> | <Star-Join Directives> | <Optimization-Goal Directives> | <Explain-Mode Directives> | <Statement cache directive> }"

"[{}| */ }]"

Usage

Use one or more optimizer directives to partially or fully specify the query plan of the optimizer. The scope of the directive is the current query only.

Directives are enabled by default. To obtain information about how specified directives are processed, view the output of the SET EXPLAIN statement. To disable directives, set the **IFX_DIRECTIVES** environment variable to `0`, or set the DIRECTIVES parameter in the ONCONFIG file to `0`.

The syntax diagram above is simplified, and does not show that the closing comment indicator must follow the same comment style as the opening comment indicator. For more information, see Optimizer Directives as Comments on page 1247.

Optimizer Directives as Comments

Optimizer directives require valid comment indicators as delimiters.

The closing delimiter you use depends on the opening delimiter:

- If `{` is the opening delimiter, you must use `}` as the closing delimiter.
- If `/*` are the opening delimiters, you must use `*/` as the closing delimiters.
- If `--` are the opening delimiters, then no closing delimiter is needed.

An optimizer directive or a list of optimizer directives immediately follows the DELETE, SELECT, or UPDATE keyword in the form of a comment. After the comment symbol, the first character in an optimizer directive is always a plus (+) sign. No blank space or other white-space character is allowed between the comment indicator and the plus sign.

You can use any of the following comment indicators:

- A double hyphen (--) delimiter

  The double hyphen needs no closing symbol because it specifies only the remainder of the current line as comment. When you use this style, include the optimizer directive on only the current line.

- Braces (**{ . . . }**) delimiters

  The comment extends from the left brace (**{**) until the next right (}) brace; this can be in the same line or in some subsequent line.

- C-language style slash and asterisk (`/* . . . */`) delimiters

  The comment extends from the initial slash-asterisk (`/*`) pair until the next asterisk-slash (`*/`) characters in the same line or in some subsequent line.

  In , the **-keepccomment** command option to the **esql** compiler must be specified when you use C-style comments.

For additional information, see How to Enter SQL Comments on page 6.

If you specify multiple directives in the same query, you must separate them with a blank space, a comma, or by any character that you choose. It is recommended that you separate successive directives with a comma.

If the query declares an alias for a table, use the alias (rather than the actual table name) in the optimizer directive specification. Because system-generated index names begin with a blank character, use quotation marks to delimit such names.

Syntax errors in an optimizer directive do not cause a valid query to fail. You can use the SET EXPLAIN statement to obtain information related to such errors.

In distributed queries, optimizer directives can reference objects in other databases of the same server instance by using the *database*:*table* or *database*:*owner*.*table* notation to qualify the name of a table in another database of the local database server.

Restrictions on Optimizer Directives

You can specify optimizer directives for any query in a DELETE, SELECT, or UPDATE statement, unless it includes any of the following syntax elements:

- A query accessing a table in a database of a remote database server instance
- In , a statement with the WHERE CURRENT OF *cursor* clause

For queries that use ANSI/ISO-compliant syntax to specify a join, the query optimizer does not follow some directives:

- The join-method directives (USE_NL, AVOID_NL, USE_HASH, AVOID_HASH, /BUILD, and /PROBE) are ignored, except in cases where the optimizer rewrites the query so that it is no longer uses the ANSI/ISO syntax.
- The join-order directive (ORDERED) is ignored in ANSI-compliant joined queries that specify the RIGHT OUTER JOIN or FULL OUTER JOIN keywords.

Access-Method Directives

Use the access-method directives to specify the manner in which the optimizer should search the tables.

Access-Method Directives

```
" { { { INDEX_ALL | MULTI_INDEX } | INDEX } ( <Table Reference> [ { | index | "index" } ] | { AVOID_INDEX |
AVOID_INDEX_SJ | INDEX_SJ } ( <Table Reference> { | index | "index" } | { FULL | AVOID_FULL | AVOID_MULTI_INDEX }
( <Table Reference> } "
```

```
" ) "
```

```
" [ comments ] "
```

Table Reference

```
" { alias | synonym | table } "
```

Element	Description	Restrictions	Syntax
alias	Temporary alternative table name declared in the FROM clause	If an *alias* is declared, it must be used (rather than *table* or *synonym*)	Identifier on page 1231
comments	Optional text that documents the directive	Must be outside the parentheses but inside the comment symbols	Character string
index	Index for which to specify the directive	Must exist. With AVOID_INDEX, AVOID_INDEX_SJ, and INDEX_SJ, at least one *index* is required	Identifier on page 1231
synonym, *table*	Name or synonym of a table to which the directive applies	Synonym and the table to which it points must exist	Identifier on page 1231

Use commas or blank spaces to separate elements within the parentheses.

The following table describes each of the access-method directives and indicates how it affects the query plan of the optimizer.

Keywords	Effect	Optimizer Action
AVOID_FULL	No full-table scan on the listed table	The optimizer considers the various indexes it can scan. If no index exists, the optimizer performs a full-table scan.

Keywords	Effect	Optimizer Action
AVOID_INDEX	Does not use any of the specified indexes	The optimizer considers the remaining indexes and a full-table scan. If all indexes for a table are specified, optimizer uses a full-table scan to access the table.
AVOID_INDEX_SJ	Does not use an index self-join path for the specified indexes	The optimizer does not consider the specified index for scanning the table in an index self-join path.
AVOID_MULTI_INDEX	Does not use a multi-index scan path for the specified table	The optimizer does not consider a multi-index scan path for the specified table.
FULL	Performs a full-table scan	Even if an index exists on a column, the optimizer uses a full-table scan to access the table.
INDEX	Uses the index specified to access the table	If more than one index is specified, the optimizer chooses the index that yields the least cost. If no indexes are specified, then all the available indexes are considered.
INDEX_ALL *or* MULTI_INDEX	Access the table using the specified indexes (Multi-index scan)	These keywords are synonyms. For usage information, see "Multi-index scans" below.
INDEX_SJ	Use the specified index to scan the table in an index self-join path.	The optimizer is forced to scan the table using an index self-join path with the specified index (or to choose the least costly index in a list of indexes for an index self-join path).

Both the AVOID_FULL and INDEX keywords specify that the optimizer should avoid a full scan of a table. It is recommended, however, that you use the AVOID_FULL keyword to specify the intent to avoid a full scan on the table.

The AVOID_MULTI_INDEX directive does not accept a list of indexes as its argument. This is because the AVOID_INDEX directive also prevents the specified index from being used in a multi-index scan execution path.

Multi-index scans

Up to sixteen (16) indexes can be defined on a table. A search path based on an access method that uses more than one index on the same table is called a *multi-index scan*. The MULTI_INDEX or INDEX_ALL directive forces the query optimizer to consider a multi-index scan to search the specified table for qualifying rows. The argument list for the MULTI_INDEX or INDEX_ALL directive has these semantics:

- If you specify a table as the only argument to the directive, the optimizer considers all of the available indexes on that table, and uses all of them (or a subset) when it searches the table for qualifying rows.
- If you specify a table and only a single index, the optimizer considers using only that index to scan the table.
- If you specify a table and more than one index, the optimizer considers a search path that uses all of the specified indexes.

Multi-index scan with skip-scan access methods

A multi-index scan path accesses a table by a *skip-scan* access method, using a sorted list of ROWIDs. The sorted list is typically generated from a multi-index scan access method, using all of the indexes that the INDEX_ALL or MULTI_INDEX directive specifies.

For example, if the query predicates specify `col1 <= 10` and `col2 BETWEEN 15 AND 25`, then the execution plan can use two indexes: the first index on **col1**, and the second index on **col2**. Each index scan returns all ROWIDs that satisfy the search condition for the respective index. The logical intersection of the two lists of ROWIDs includes only the rows that satisfy both search conditions. The database server then sorts the combined ROWID list, and uses this sorted list to scan the table for the result set of the query.

If the query includes predicates on more than two indexed columns, the list of ROWIDs that each index scan returns must be combined to produce a sorted ROWID list of all the qualifying rows.

Because each ROWID represents the physical location of a row (on which page and in which slot), the execution path simply accesses that physical location to retrieve the row. As the term "skip-scan" suggests, there are typically gaps from one ROWID to the next in the sorted list, so that the database server "skips" from one qualifying row to the next qualifying row of the result set.

The list of sorted ROWIDs can be generated from multiple index scans, as described above, or from a single index scan. In the case of a single index, the skip-scan execution path takes these actions:

1. The single index scan creates an unsorted list of the ROWIDs of all qualifying rows.
2. This unsorted list is sorted by ROWID value.
3. The database server then retrieves the qualifying rows in the order of their ROWIDs.

A skip-scan access method resembles a sequential scan, but can sometimes be more efficient. A sequential scan retrieves every row in the table, but a skip-scan only retrieves the rows that have qualifying ROWIDs.

Restrictions on multi-index scan paths for query execution

The transaction isolation level affects whether the MULTI_INDEX or INDEX_ALL directive can force a multi-index scan execution path, which is not available while the isolation level is Cursor Stability, or is Committed Read with the LAST COMMITTED option. (This directive is supported, however, in the Dirty Read and Repeatable Read isolation levels, and in Committed Read without the LAST COMMITTED option.)

The following additional restrictions apply to multi-index scan access paths:

- The indexes must be B-tree indexes. These can be attached or detached indexes.
- These directives are ignored for R-tree indexes, functional indexes, and indexes based on the Virtual Index Interface (VII).
- The table cannot be a remote table, a pseudo-table, a system catalog table, an external table, or a hierarchical table.
- A multi-index scan cannot support join predicates as index filters in the underlying index scans.
- A multi-index scan ignores all columns of a composite index except the leading column.
- DML statements that perform cascade deletes or declare statement local variables (SLVs) cannot use a multi-index scan.
- Update queries that activate a FOR EACH ROW triggered action cannot use a multi-index scan.
- In ANSI-compliant databases, the MULTI_INDEX or INDEX_ALL directive is not followed for a SELECT statement that has no ORDER BY clause, no GROUP BY clause, and no FOR READ ONLY clause, if the FROM clause specifies only a single table. (In this special case, the query has implicit cursor behavior that conflicts with a multi-index scan access path.)

Combinations of access method directives

In general, you can specify only one access-method directive per table. Only the following combinations of access-method directives are valid for the same table in the same query:

- INDEX, AVOID_INDEX_SJ
- AVOID_FULL, AVOID_INDEX
- AVOID_FULL, AVOID_INDEX_SJ
- AVOID_INDEX, AVOID_INDEX_SJ
- AVOID_FULL, AVOID_INDEX, AVOID_INDEX_SJ
- AVOID_FULL, AVOID_MULTI_INDEX
- AVOID_INDEX, AVOID_MULTI_INDEX
- AVOID_INDEX_SJ, AVOID_MULTI_INDEX
- AVOID_FULL, AVOID_INDEX_SJ, AVOID_MULTI_INDEX
- AVOID_INDEX, AVOID_INDEX_SJ, AVOID_MULTI_INDEX

When you specify both the AVOID_FULL and AVOID_INDEX access-method directives, the optimizer avoids performing a full scan of the table and it avoids using the specified index or indexes. This combination of negative directives allows the optimizer to use indexes that are created after the access-method directives are specified.

Because the optimizer automatically considers the index self-join path if you specify the INDEX or AVOID_FULL directive, use the INDEX_SJ directive only to force an index self-join path using the specified index (or choosing the least costly index in a comma-separated list of indexes). The INDEX_SJ directive can improve performance when a multicolumn index includes columns that provide only low selectivity as index key filters.

Specifying the INDEX_SJ directive circumvents the usual optimizer requirement for data distribution statistics on the lead keys of the index. This directive causes the optimizer to consider an index self-join path, even if data distribution statistics are not available for the leading index key columns. In this case, the optimizer only includes the minimum number of index key columns as lead keys to satisfy the directive.

For example, if an index is defined on columns **c1**, **c2**, **c3**, **c4**, and the query specifies filters on all four of these columns but no data distributions are available on any column, then specifying INDEX_SJ on this index will result in column **c1** being used as the lead key in an index self-join path. If you want the optimizer to use an index but not to consider the index self-join path, then you must specify an INDEX or AVOID_FULL directive to choose the index, and you must also specify an AVOID_INDEX_SJ directive to prevent the optimizer from considering any other index self-join path.

If AVOID_INDEX_SJ is used together with the INDEX directive, either as an explicit INDEX directive or as the equivalent AVOID_FULL and AVOID_INDEX combination, the indexes specified in the AVOID_INDEX_SJ directive must be a subset of the indexes specified in the INDEX directive. For more information about the effects of the INDEX_SJ and AVOID_INDEX_SJ directives, see the chapter of the that describes optimizer directives.

Specifying the MULTI_INDEX or INDEX_ALL directive circumvents the usual optimizer requirement for statistics on the specified table. The optimizer normally requires at least low level statistics on the table before considering multi-index scan path on the table.

Examples of Access Method Directives

Suppose that you have a table named **emp** that contains the columns **emp_no**, **dept_no**, and **job_no**, and for which the following indexes **ids_dept_no** index is defined on the **dept_no** column, and the **idx_job_no** index is defined on the **job_no** column. When you perform a SELECT that includes the **emp** table in the FROM clause, you might direct the optimizer to access the table in one of the following ways:

- Example using a positive directive:

  ```
  SELECT {+INDEX(emp idx_dept_no)} ...
  ```

  In the example above, the access-method directive forces the optimizer to consider an execution path that scans the **idx_dept_no** index on the **dept_no** column.

  In the following example the access-method directive forces the optimizer to consider using a multi-index scan, based on the combined results of scanning both the **idx_dept_no** index on the **dept_no** column and the **idx_job_no** index on the **job_no** column.

  ```
  SELECT {+MULTI_INDEX(emp idx_dept_no ids_job_no)} ...
  ```

- Example using negative directives:

  ```
  SELECT {+AVOID_INDEX(emp idx_loc_no, idx_job_no), AVOID_FULL(emp)} ...
  ```

  This example includes multiple access-method directives. These directives force a scan of the **idx_dept_no** index on the **dept_no** column by instructing the optimizer not to scan the **idx_loc_no** and **idx_job_no** indexes, and not to perform a full scan of the **emp** table. If a new **idx_emp_no** index, however, is created for table **emp**, these directives do not prevent the optimizer from considering it.

Note also that the term *negative directive* refers to the string "AVOID_" in an access method directive, and has nothing to do with the + symbol following the comment indicator that begins every optimizer directive.

Join-Order Directive

Use the ORDERED join-order directive to force the optimizer to join tables or views in the order in which they are referenced in the FROM clause of the query.

Join-Order Directive

" ORDERED "

" [*comments*] "

Element	Description	Restrictions	Syntax
comments	Text to document the directive	Must appear between comment symbols	Character string

For example, the following query forces the database server to join the **dept** and **job** tables and then join the result with the **emp** table:

```
SELECT --+ ORDERED
   name, title, salary, dname
   FROM dept, job, emp WHERE title = 'clerk' AND loc = 'Palo Alto'
   AND emp.dno = dept.dno
   AND emp.job= job.job;
```

Because no predicates occur between the **dept** table and the **job** table, this query forces the database server to construct a Cartesian product.

When your query involves a view, the placement of the ORDERED join-order directive determines whether you are specifying a partial- or total-join order.

- Specifying partial-join order when you create a view

  If you use the ORDERED directive when you create a view, the base tables are joined contiguously in the order of the view definition.

  For all subsequent queries on the view, the database server joins the base tables contiguously in the order specified in the view definition. When used in a view, the ORDERED directive does not affect the join order of other tables named in the FROM clause in a query.

- Specifying total-join order when you query a view

  When you specify the ORDERED join-order directive in a query that uses a view, all tables are joined in the order specified, even those tables that form views. If a view is included in the query, the base tables are joined contiguously in the order of the view definition. For examples of ORDERED with views, refer to your .

Because of ordering requirements for OUTER joins, in ANSI-compliant joined queries that specify the RIGHT OUTER JOIN or FULL OUTER JOIN keywords, the ORDERED join-order directive is ignored, but it is listed under *Directives Not Followed* in the explain output file.

Join-Method Directives

Use join-method directives to influence how tables are joined in the Informix®-extension joined query.

(explicit id unique_1725_Connect_42_jmdi003) unique_1725_Connect_42_jmdi003

Join-Method Directives

" { { **USE_NL | AVOID_NL** } **(** <Table Reference>[] | { **AVOID_HASH | USE_HASH** } **(** <Table Reference>[] [{ **/BUILD | /PROBE** }] } "

" **)** "

" [*comments*] "

Element	Description	Restrictions	Syntax
comments	Text to documents the directive	Must appear between comment symbols	Character string

Use commas or blank spaces to separate the elements within the parentheses.

The following table describes each of the join-method directives.

Keyword

Effect

USE_NL

Uses the specified tables as the inner table in a nested-loop join

If *n* tables are specified in the FROM clause, then at most (*n*-1) tables can be specified in the USE_NL join-method directive.

USE_HASH

Uses a hash join to access the specified table

You can also choose whether the table will be used to create the hash table or to probe the hash table.

AVOID_NL

Does not use the specified table as inner table in a nested loop join

A table listed with this directive can still participate in a nested loop join as the outer table.

AVOID_HASH

Does not access the specified table using a hash join

You can optionally use a hash join, but impose restrictions on the role of the table within the hash join.

A join-method directive takes precedence over the join method forced by the OPTCOMPIND configuration parameter.

When you specify the USE_HASH or AVOID_HASH directives (to use or avoid a hash join, respectively), you can also specify the role of each table:

- /BUILD

  With the USE_HASH directive, this keyword indicates that the specified table be used to construct a hash table. With the AVOID_HASH directive, this keyword indicates that the specified table *not* be used to construct a hash table.

- /PROBE

  With the USE_HASH directive, this keyword indicates that the specified table be used to probe the hash table. With the AVOID_HASH directive, this keyword indicates that the specified table *not* be used to probe the hash table. You can specify multiple probe tables as long as there is at least one table for which you do not specify PROBE.

For the optimizer to find an efficient join query plan, you must at least run UPDATE STATISTICS LOW for every table that is involved in the join, so as to provide appropriate cost estimates. Otherwise, the optimizer might choose to broadcast the entire table to all instances, even if the table is large.

If neither the /BUILD nor the /PROBE keyword is specified, the optimizer uses cost estimates to determine the role of the table.

In this example, the USE_HASH directive forces the optimizer to construct a hash table on the **dept** table and consider only the hash table to join **dept** with the other tables. Because no other directives are specified, the optimizer can choose the least expensive join methods for the other joins in the query.

```
SELECT /*+ USE_HASH (dept /BUILD)
   The optimizer must use dept to construct a hash table */
   name, title, salary, dname
   FROM emp, dept, job WHERE loc = 'Phoenix'
      AND emp.dno = dept.dno  AND emp.job = job.job;
```

Join-method optimizer directives that you specify for an ANSI-compliant joined query are ignored, but they are listed under *Directives Not Followed* in the explain output file.

Star-Join Directives

Use the star-join directives to specify the manner in which the optimizer should join tables that have a star schema.

Star-Join Directives

"{ **AVOID_FACT**({ <Table Reference>}) | **FACT**(<Table Reference>) | **STAR_JOIN** | **AVOID_STAR_JOIN** }"

"[*comments*]"

Table Reference

"{ *alias* | *synonym* | *table* }"

Element	Description	Restrictions	Syntax
alias	Temporary alternative table name declared in the FROM clause	If an *alias* is declared, it must be used (rather than *table* or *synonym*)	Identifier on page 1231
comments	Optional text that documents the directive	Must be outside the parentheses but inside the comment symbols	Character string
synonym, *table*	Name or synonym of a table to which the directive applies	Synonym and the table to which it points must exist	Identifier on page 1231

In AVOID_FACT directives that specify more than one table, use a comma or blank space to separate consecutive elements within the parentheses.

The following table describes each of the star-join directives and indicates how it affects the query plan of the optimizer.

Keywords	Effect	Optimizer Action
AVOID_FACT	At least one table must be specified. Do not use the table (or any table in the list of tables) as a fact table in star-join optimization.	The optimizer does not consider a star-join execution plan that treats the specified table (or any of the tables in the list of tables) as a fact table.
AVOID_STAR_JOIN	The optimizer does not consider a star-join execution plan.	The optimizer chooses a query execution plan that is not a star-join plan.
FACT	Exactly one table must be specified. Only consider the specified table as a fact table in the star-join execution plan.	These optimizer considers a query plan in which the specified table is a fact table in a star-join execution plan.

Keywords	Effect	Optimizer Action
STAR_JOIN	Favor a star-join plan, if one is possible.	The optimizer favors a star-join execution plan, if available.

The star-join directives require that the parallel database query feature (PDQ) be enabled. Star join query optimization is disabled when PDQ is off.

The star-join directives require that all tables in the query have at least low level statistics. If table statistics are not available for any table in the query, star-join query optimization is disabled.

The SET OPTIMIZATION ENVIRONMENT STAR_JOIN DISABLED statement of SQL disables star-join optimization in the current session. (For additional information about optimization environment settings, see ENVIRONMENT Options on page 823.)

Specifying the FACT directive alone does not automatically favor a star-join execution plan. You can direct the optimizer to prefer a star-join execution plan with a specific fact table by specifying a combination of a STAR_JOIN directive and a FACT directive.

You can view the star join optimization path of a query in the output file of the SET EXPLAIN statement, or by using to obtain Visual Explain output.

In cluster environments, the star-join optimizer directives are valid on these types of secondary servers:

- Shared disk secondary servers (SDS)
- Remote standalone secondary servers (RSS)
- High-availability data replication secondary servers (HDR).

Restrictions on star-join directives

The following restrictions apply to queries that attempt to join tables that have star-schema dependencies:

- The parallel database query (PDQ) feature must be enabled for star-join directives to be valid.
- All tables in the query must have at least low level statistics.
- Star-join directives do not support joins of more than one fact table.
- Star-join directives are not valid while the transaction isolation level is Committed Read Last Committed or Cursor Stability. (All other transaction isolation levels are supported.)

Optimization-Goal Directives

Use optimization-goal directives to specify the measure that is used to determine the performance of a query result.

Optimization-Goal Directives

" { **ALL_ROWS | FIRST_ROWS** } "

" [*comments*] "

Element	Description	Restrictions	Syntax
comments	Text documenting the directive	Must appear between comment symbols	Character string

The two optimization-goal directives are:

- FIRST_ROWS

  This tells the optimizer to choose a plan that optimizes the process of finding only the first screenful of rows that satisfies the query. Use this option to decrease initial response time for queries that use an interactive mode or that require the return of only a few rows.

- ALL_ROWS

  This directive tells the optimizer to choose a plan that optimizes the process of finding all rows that satisfy the query.

  This form of optimization is the default.

An optimization-goal directive takes precedence over the **OPT_GOAL** environment variable setting and over the OPT_GOAL configuration parameter.

For information about how to set the optimization goal for an entire session, see the SET OPTIMIZATION statement.

You cannot use an optimization-goal directive in the following contexts:

- In a view definition
- In a subquery

The following query returns the names of the employees who earned the top fifty bonuses. The optimization-goal directive directs the optimizer to return the first screenful of rows as fast as possible.

```
SELECT {+FIRST_ROWS
   Return the first screenful of rows as fast as possible}
   LIMIT 50 fname, lname  FROM employees ORDER BY bonus DESC;
```

Explain-Mode Directives

Use the explain-mode directives to test and debug query plans and to print information about the query plan to the explain output file.

Explain-Mode Directives

" EXPLAIN "

" [, AVOID_EXECUTE] "

" [*comments*] "

Element	Description	Restrictions	Syntax
comments	Text documenting the directive	Must appear between comment symbols	Character string

The following table lists the effect of each explain-mode directive.

Keyword

Effect

EXPLAIN

Turns SET EXPLAIN ON for the specified query

AVOID_EXECUTE

Prevents the data manipulation statement from executing; instead, the query plan is printed to the explain output file

The EXPLAIN directive is primarily useful for testing and debugging query plans. It is redundant when SET EXPLAIN ON is already in effect. It is not valid in a view definition or in a subquery.

The next query executes and prints the query plan to the explain output file:

```
SELECT {+EXPLAIN}
   c.customer_num, c.lname, o.order_date
   FROM customer c, orders o WHERE c.customer_num = o.customer_num;
```

The AVOID_EXECUTE directive prevents execution of a query on either the local or remote site, if a remote table is part of the query. This directive does not prevent nonvariant functions in a query from being evaluated.

The next query does returns no data, but writes its query plan to the explain output file:

```
SELECT {+EXPLAIN, AVOID_EXECUTE}   c.customer_num, c.lname, o.order_date
   FROM customer c, orders o WHERE c.customer_num = o.customer_num;
```

You must use both the EXPLAIN and AVOID_EXECUTE directives to see the query plan of the optimizer (in the explain output file) without executing the query. The comma (,) separating these two directives is optional.

If you omit the EXPLAIN directive when you specify the AVOID_EXECUTE directive, no error is issued, but no query plan is written to the explain output file and no DML statement is executed.

You cannot use the explain-mode directives in the following contexts:

- In a view definition
- In a trigger
- In a subquery

They are valid, however, in a SELECT statement within an INSERT statement.

Statement cache directive

Use the statement cache directive to prevent the statement from being stored in the statement cache. The AVOID_STMT_CACHE optimizer directive forces the optimizer to reoptimize the statement every time that the statement is run.

Statement cache directive

" AVOID_STMT_CACHE "

" [*comments*] "

Element	Description	Restrictions	Syntax
comments	Text documenting the directive	Must appear between comment symbols	Character string

The AVOID_STMT_CACHE directive is useful for those statements whose query plans can change significantly depending on the values of the placeholders that are passed to the database server when the statements are run. Using a cached query plan for those sorts of statements might result in poor performance.

For example, the following statement is not stored in the statement cache:

```
SELECT {+AVOID_STMT_CACHE}
    c.customer_num, c.lname, o.order_date
    FROM customer c, order o
    WHERE c.customer_num = o.customer_num;
```

External Directives

You can use the SAVE EXTERNAL DIRECTIVES statement to store optimizer directives in the **sysdirectives** table of the system catalog. Informix® applies these external directives automatically to subsequent queries and subqueries that match a specified SELECT statement.

The EXT_DIRECTIVES configuration parameter and the **IFX_EXTDIRECTIVES** environment variable can be set to control whether external directives are enabled or disabled for the database server instance or for the session. Setting either of these to zero disables external directives; setting both to `1` enables external directives.

You can also use the EXTDIRECTIVES option of the SET ENVIRONMENT statement to enable or disable external directives during a session. For more information, see Enabling or disabling external directives for a session on page 686.

Owner name

The owner name specifies the owner of a database object. Use this segment whenever you see a reference to Owner Name in a syntax diagram.

Syntax

(explicit id unique_42_Connect_42_own001) unique_42_Connect_42_own001

Owner Name

" { *"owner"* | *owner* | *'owner'* } "

Element	Description	Restrictions	Syntax
owner	User name of the owner of an object in a database	Maximum length is 32 bytes	Must conform to the rules of your operating system.

Usage

In an ANSI-compliant database, you must specify the *owner* of any database object that you do not own. In reference to the owner of database objects, the ANSI/ISO synonym for owner name is *authorization identifier*. (In reference to schema objects, however, the ANSI/ISO term for what the Informix® documentation calls an *owner name* is *schema name*.)

In databases that are not ANSI-compliant, the *owner* name is optional. You do not need to specify *owner* when you create database objects or use data access statements. If you do not specify *owner* when you create a database object, the database server assigns your login name as the owner of the object, in most cases. For exceptions to this rule, see Ownership of Created Database Objects on page 235 in the CREATE FUNCTION statement description, and Ownership of Created Database Objects on page 281 in the CREATE PROCEDURE statement description. When a DDL statement in an owner-privileged UDR creates a new database object, the owner of the routine (rather than the user who executes it, if that user is not the owner of the routine) becomes the owner of the new database object.

If you specify *owner* in data-access statements, the database server checks it for correctness. Without quotation marks, *owner* is case insensitive. The following four queries all can access data from the table **kaths.tab1**:

```
SELECT * FROM tab1;
SELECT * FROM kaths.tab1;
SELECT * FROM KATHS.tab1;
SELECT * FROM Kaths.tab1;
```

In an ANSI-compliant database, only the owner of the table, user **kaths**, can issue the first of these example queries, which specifies an unqualified table name, but any user who holds the Select privilege on **tab1** can issue that query in a database that is not ANSI-compliant. For more information about owner names in ANSI-compliant databases, see ANSI-Compliant Database Restrictions and Case Sensitivity on page 1264.

A *role* that the CREATE ROLE statement declares is an authorization identifier, and is therefore subject to the syntax restrictions on owner names, but a role cannot be the owner of a database object. Similarly, the keyword PUBLIC, which specifies the group of all users, cannot be the owner of a database object, except in the special cases of the **sysdbopen( )** and **sysdbclose( )** procedures. For more information about these built-in session configuration UDRs, see Session Configuration Procedures.

Using Quotation Marks

Within quotation marks, *owner* is case sensitive. Quotation marks instruct the database server to read or store the name exactly as typed when you create or access a database object. For example, suppose that you have a table whose owner is `Sam`. You can use either one of the following two statements to access data in the table:

```
SELECT * FROM table1;
SELECT * FROM 'Sam'.table1;
```

The first query succeeds because the owner name is not required. The second query succeeds because the specified owner name matches the owner name as it is stored in the database.

Referencing Tables Owned by User informix

If you use the owner name as one of the selection criteria to access database object information from one of the system catalog tables, the owner name is case sensitive. To preserve lettercase, you must enclose *owner* in single or double quotation marks, and you must type the owner name exactly as it is stored in the system catalog table. Of the following two examples, only the second successfully accesses information on the table **Kaths.table1**.

```
SELECT * FROM systables WHERE tabname = 'tab1' AND owner = 'kaths';
SELECT * FROM systables WHERE tabname = 'tab1' AND owner = 'Kaths';
```

User **informix** is the owner of the system catalog tables, and in an ANSI-compliant database you must specify **informix** as a qualifier when SQL statements reference system catalog tables, unless you are user **informix**:

```
SELECT * FROM "informix".systables WHERE tabname = 'tab1' AND owner = 'Kaths';
```

Informix® accepts any of the following notations to specify a system catalog table of an ANSI-compliant database:

- `"informix".`*system_table*
- `informix.`*system_table*
- `'informix'.`*system_table*

Of these three formats, however, only the first, where the *owner* is specified as a delimited identifier, is directly interoperable with most other database servers. For the format with no delimiters, the ANSI/ISO standard for SQL upshifts the lowercase letters to `INFORMIX`, and the same standard does not support single (') quotation marks as valid delimiters for owner names or for schema names.

In contrast, Informix® treats the name **informix** as a special case, and preserves lowercase letters when **informix** is specified, with or without delimiters, whether or not the database is ANSI-compliant. To write SQL code that is portable to non-Informix® database servers, however, you should always delimit the owner names of database objects between double (`"`) quotation marks.

The following SQL examples use undelimited owner names:

```
CREATE TABLE informix.t1(i SERIAL NOT NULL);
CREATE TABLE someone.t1(i SERIAL NOT NULL);
```

If these statements execute successfully, the first table has `informix` registered in **systables** as the owner, and the second has `SOMEONE` registered as the owner. When the owner name is delimited by quotation marks in SQL statements, the specified lettercase of *owner* is preserved, but the lettercase does not matter when the owner name is undelimited, because Informix® upshifts most undelimited owner names, but downshifts the undelimited **informix** (or **INFORMIX**) owner name to `informix`.

For example, suppose that after the previous two CREATE TABLE statements execute successfully, user **informix** issues the following statement:

```
CREATE TABLE INFORMIX.t1(i SERIAL NOT NULL);
```

This statement fails, because the combination of *owner* name and *table* name is not unique, if the previously registered table **t1** that is owned by **informix** already exists in the database.

Tip: The USER operator returns the login name of the current user exactly as it is stored on the system. If the owner name is stored differently from the login name (for example, a mixed-case owner name and an all lowercase login name), the `owner = USER` syntax fails.

ANSI-Compliant Database Restrictions and Case Sensitivity

The following table describes how the database server reads and stores *owner* when you create, rename, or access a database object.

Owner Name Specification	What the ANSI-Compliant Database Server Does
Omitted	Reads or stores *owner* exactly as the login name is stored in the system, but returns an error if the user is not the *owner*.
Specified without quotation marks	Reads or stores *owner* in uppercase letters
Enclosed between quotation marks	Reads or stores *owner* exactly as entered. See also Using Quotation Marks on page 1263 and Referencing Tables Owned by User informix on page 1263.

If you specify the owner name when you create or rename a database object in an ANSI-compliant database, you must include the owner name in data access statements. You must include the owner name when you access a database object that you do not own.

Because the database server automatically shifts *owner* to uppercase letters if not between quotation marks, case-sensitive errors can cause queries to fail. For example, if you are user **nancy** and you use the following statement, the resulting view has the name **nancy.njcust**:

```
CREATE VIEW 'nancy'.njcust AS
   SELECT fname, lname FROM customer WHERE state = 'NJ';
```

The following SELECT statement fails because it tries to match the name **NANCY.njcust** to the actual owner and table name of **nancy.njcust**:

```
SELECT * FROM nancy.njcust;
```

In a distributed query, if the owner name is not between quotation marks, the remote database follows the lettercase convention of the local database. If the local database is ANSI-compliant, then the remote database processes the owner name in *uppercase*. If the local database is not ANSI compliant, then the remote database processes the owner name in *lowercase*.

Tip: When you use the owner name as one of the selection criteria in a query (for example, `WHERE owner = 'kaths'`), make sure that the quoted string matches the owner name exactly as it is stored in the database. If the database server cannot find the database object or database, you might need to modify the query so that the quoted string uses uppercase letters (for example, `WHERE owner = 'KATHS'`).

Because owner name is an authorization identifier, rather than an SQL identifier, you can enclose *owner* between single-quotation marks (') in SQL statements of a database where the **DELIMIDENT** environment variable specifies support for delimited identifiers, thereby requiring double-quotation marks (") around SQL identifiers.

Setting ANSIOWNER for an ANSI-Compliant Database

About this task

The default behavior of an ANSI-compliant database is to replace any lowercase letters with uppercase letters in any *owner* specification that is not enclosed in quotation marks. You can prevent this by setting the **ANSIOWNER** environment variable to 1 before the database server is initialized. This preserves whatever lettercase you use when you specify the *owner* string without quotation marks.

Default Owner Names

If you create a database object without explicitly specifying an owner name in a database that is not ANSI-compliant, your authorization identifier (as the default owner of the object) is stored in the system catalog of the database as if you had specified your authorization identifier within quotation marks (that is, preserving the lettercase).

If you create a database object without explicitly specifying an owner name in a database that is ANSI-compliant, any lowercase letters in your authorization identifier (as the default owner of the object) are stored in the system catalog of the database in uppercase characters, unless the **ANSIOWNER** environment variable was set to 1 before the database server was initialized. If **ANSIOWNER** was set to 1, however, the database stores the default owner of the object as your authorization identifier, with its lettercase preserved.

Summary of Lettercase Rules for Owner Names

To create a database object, such as a table called **mytab**, a user whose login name is **Otho** can declare the name of the new database object in any of the following ways:

1. `CREATE TABLE mytab . . .`
2. `CREATE TABLE Otho.mytab . . .`
3. `CREATE TABLE "Otho".mytab . . .`

The format in which an undelimited *owner* name (as in the second example) is stored in the **owner** column of the **systables** system catalog table is dependent on whether or not the local database is an ANSI-compliant database.

- In case 1, no *owner* name is specified. The implicit owner of the table is **Otho**, the user who created the table, and that owner name is stored in the **systables** table in the same format (**Otho**) as the user ID of the owner, independent of the ANSI-compliance status of the database.
- In case 2, an undelimited *owner* name is specified. The **systables** table stores all letters in the *owner* name in lowercase (here as **otho**) for databases that are not ANSI-compliant databases. For ANSI-compliant databases in which **ANSIOWNER** is not set to `1`, **systables** table stores all *owner* name letters in uppercase (here as **OTHO**). If **ANSIOWNER** is set to `1`, however, the name is stored in the same lettercase as specified in the DDL statement (here as **Otho**).
- In case 3, the delimited *owner* name is stored in the **systables** table in the same format in which it was specified (here as **Otho**), independent of the ANSI-compliance status of the database.

Note that user identifiers are case sensitive, but database object names are case insensitive. Therefore, the same user cannot own both a table **tab** and a table **TAB**.

In addition to the CREATE TABLE statement in these examples, all SQL statements and SPL statements follow these rules where a table name can be specified. For example, when using DROP TABLE, the format in which owner name appears while the statement is being processed is dependent upon the same conditions:

- whether an explicit owner name is specified.
- if an explicit owner name is specified, whether quotation marks delimit the owner name.
- if an explicit owner name is not delimited by quotation marks, whether or not the database is ANSI compliant.
- if the database is ANSI compliant, whether or not **ANSIOWNER** was set to `1` before the database was initialized.

Purpose Options

The CREATE ACCESS_METHOD, CREATE XADATASOURCE TYPE, and ALTER ACCESS_METHOD statements of Informix® can specify purpose options for user-defined routines with the following syntax.

Syntax

Purpose Options

" { *task* = *external_routine* | *value* = { *string_value* | *numeric_value* } | *flag* } "

Element	Description	Restrictions	Syntax
external_routine	User-defined routine that performs a *task*	Must be registered in the database	Database Object Name on page 1225
flag	Keyword indicating which feature a flag enables	The interface specifies flag names	*Flag* Purpose Category in the table in Purpose Functions, Flags, and Values on page 1267.
numeric_value	A value of a real number	Must be within the range of a numeric data type	Literal Number on page 1198
string_value	A value that is expressed as one or more characters	Characters must be from the code set of the database	Quoted String on page 1202.
task	Keyword that identifies a purpose function	Keywords to which you can assign a function (whose name cannot match the keyword)	*Task* Purpose Category in the table in Purpose Functions, Flags, and Values on page 1267.
value	Keyword that identifies configuration information	Predefined configuration keywords to which you can assign values	*Value* Purpose Category in the table in Purpose Functions, Flags, and Values on page 1267.

Usage

Informix® supports purpose options in two contexts:

- Defining or modifying primary and secondary access methods for local or remote tables, views, and indexes
- Defining access methods for XA-compliant external data sources.

Purpose Functions, Flags, and Values

Purpose functions, methods, and flags defined the attributes of access methods.

The following table describes the possible settings for the **sysams** columns that contain purpose functions or methods, flags, and values. The entries appear in the same order as the corresponding **sysams** columns.

Table 30. Purpose functions, purpose flags, and purpose values

Keyword	Explanation	Category	Default
am_sptype	A character that specifies from what type of storage space a primary or secondary-access method can access data. The **am_sptype** character can have any of the following settings:	Value	Virtual-Table Interface (C): 'A'

Table 30. Purpose functions, purpose flags, and purpose values (continued)

Keyword	Explanation	Category	Default
	• 'X' indicates the method accesses only extspaces. • 'S ' indicates the method accesses only sbspaces. • 'A' indicates the method can access extspaces and sbspaces. Valid only for a new access method. You cannot change or add an **am_sptype** value with ALTER ACCESS_METHOD. Do not set **am_sptype** to 'D' or attempt to store a virtual table in a dbspace.		
am_defopclass	The default operator class for a secondary-access method. The access method must exist before you can define its operator class, so you set this value in the ALTER ACCESS_METHOD statement.	Value	None
am_keyscan	A flag that, if set, indicates that **am_getnext** returns rows of index keys for a secondary-access method. If a query selects only the columns in the index key, the database server uses the row of index keys that the secondary-access method puts in shared memory, without reading the table.	Flag	Not set
am_unique	A flag to set if a secondary-access method checks for unique keys	Flag	Not set
am_cluster	A flag that you set if a primary- or secondary-access method supports clustering of tables	Flag	Not set
am_rowids	A flag that you set if a primary-access method can retrieve a row from a specified address	Flag	Not set
am_readwrite	A flag to set if a primary-access method supports data changes. The default setting, not set, indicates that the virtual data is read-only. For the C Virtual-Table Interface, set this flag if your application writes data, to avoid the following problems: • An INSERT, DELETE, UPDATE, or ALTER FRAGMENT statement causes an SQL error. • Function **am_insert**, **am_delete**, or **am_update** is not run.	Flag	Not set
am_parallel	A flag that the database server sets to indicate which purpose functions or methods can run in parallel in a primary or secondary-access method. If set, the hexadecimal **am_parallel** bitmap contains one or more of the following bit settings:	Flag	Not set

Table 30. Purpose functions, purpose flags, and purpose values (continued)

Keyword	Explanation	Category	Default
	• The `1` bit is set for parallelizable scan. • The `2` bit is set for parallelizable delete. • The `4` bit is set for parallelizable update. • The `8` bit is set for parallelizable insert. Insertions, deletions, and updates are not supported in the .		
am_expr_pushdown	A flag that enables the use of parameter descriptors.	Flag	Not set
am_costfactor	A value by which the database server multiplies the cost that the am_scancost purpose function or method returns for a primary or secondary-access method. An **am_costfactor** value from 0.1 to 0.9 reduces the cost to a fraction of the value that **am_scancost** calculates. An **am_costfactor** value of 1.1 or greater increases the **am_scancost** value.	Value	1.0
am_create	A keyword that you associate with a user-defined function or method (UDR) name that creates a virtual table or virtual index	Task	None
am_drop	A keyword that you associate with the name of a UDR that drops a virtual table or virtual index	Task	None
am_open	A keyword that you associate with the name of a UDR that makes a fragment, extspace, or sbspace available	Task	None
am_close	A keyword that you associate with the name of a UDR that reverses the initialization that **am_open** performs	Task	None
am_insert	A keyword that you associate with the name of a UDR that inserts a row or an index entry	Task	None
am_delete	A keyword that you associate with the name of a UDR that deletes a row or an index entry	Task	None
am_update	A keyword that you associate with the name of a UDR that changes the values in a row or key	Task	None
am_stats	A keyword that you associate with the name of a UDR that builds statistics based on the distribution of values in storage spaces	Task	None
am_scancost	A keyword that you associate with the name of a UDR that calculates the cost of qualifying and retrieving data	Task	None
am_check	A keyword that you associate with the name of a UDR that tests the physical structure of a table or performs an integrity check on an index	Task	None

Table 30. Purpose functions, purpose flags, and purpose values (continued)

Keyword	Explanation	Category	Default
am_beginscan	A keyword that you associate with the name of a UDR that sets up a scan	Task	None
am_endscan	A keyword that you associate with the name of a UDR that reverses the setup that **am_beginscan** initializes	Task	None
am_rescan	A keyword that you associate with the name of a UDR that scans for the next item from a previous scan to complete a join or subquery	Task	None
am_getnext	A keyword that you associate with the name of the required UDR that scans for the next item that satisfies a query	Task	None
am_getbyid	A keyword that you associate with the name of a UDR that fetches data from a specific physical address; **am_getbyid** is available only for primary-access methods	Task	None
am_truncate	A keyword that you associate with the name of a UDR that deletes all rows of a virtual table (primary-access method) or that deletes all corresponding keys in a virtual index (secondary-access method)	Task	None

The following rules apply to the purpose-option specifications in the CREATE ACCESS_METHOD and ALTER ACCESS_METHOD statements:

- To specify multiple purpose options in one statement, separate them with commas.
- The CREATE ACCESS_METHOD statement must specify a user-defined function or method name that corresponds to the **am_getnext** keyword.

  The ALTER ACCESS_METHOD statement cannot drop the function or method name that corresponds to **am_getnext** but can modify it.
- The ALTER ACCESS_METHOD statement cannot add, drop, or modify the **am_sptype** value.
- You can specify the **am_defopclass** value only with the ALTER ACCESS_METHOD statement.

  You must first register a secondary-access method with the CREATE ACCESS_METHOD statement before you can assign a default operator class.

Purpose Options for XA Data Source Types

The CREATE XADATASOURCE TYPE statement specifies purpose functions that provide access to data from external data sources that comply with the X/Open XA standards. These functions also enable external data to be processed in accordance with the transactional semantics of Informix®. Only databases that use transaction logging, such as ANSI-compliant databases and Informix® databases that support explicit transactions, can support transaction coordination.

The following statement creates a new XA data source type called **MQSeries®**, owned by user **informix**.

```
CREATE XADATASOURCE TYPE 'informix'.MQSeries(
                            xa_flags    = 1,
                            xa_version  = 0,
                            xa_open     = informix.mqseries_open,
                            xa_close    = informix.mqseries_close,
                            xa_start    = informix.mqseries_start,
                            xa_end      = informix.mqseries_end,
                            xa_rollback = informix.mqseries_rollback,
                            xa_prepare  = informix.mqseries_prepare,
                            xa_commit   = informix.mqseries_commit,
                            xa_recover  = informix.mqseries_recover,
                            xa_forget   = informix.mqseries_forget,
                           xa_complete = informix.mqseries_complete);
```

These values represent the fields in the XA Switch Structure, as listed in the file **$INFORMIXDIR/incl/public/xa.h**. The order of specifications in this example follows the order of column names in the **sysxasourcetypes** system catalog table, but they can be listed in any order, provided that no item is repeated. The **xa_flags** and **xa_version** values must be numbers; the rest must be names of UDRs that the Transaction Manager can invoke. These UDRs must already exist in the database before you can issue a CREATE XADATASOURCE TYPE statement that references them among its purpose option specifications.

The DROP FUNCTION or DROP ROUTINE statement cannot drop a UDR that is listed among the purpose options of a CREATE XADATASOURCE TYPE statement until all of the XA datasource types that were defined using the UDR are dropped.

For information about how to use the UDRs in the previous example to coordinate transactions with external XA data sources, see the *HCL® Informix® DataBlade® API Programmer's Guide*.

For information about the MQDataBlade module, see the *IBM® Informix® Database Extensions User's Guide*.

Return Clause

The Return clause specifies the data type of a value or values that a user-defined function returns. You can use this segment in UDR definitions.

Syntax

(explicit id unique_309_Connect_42_recla003) unique_309_Connect_42_recla003(explicit id unique_309_Connect_42_recla004)

unique_309_Connect_42_recla004

Return Clause

" { **RETURNING** | **RETURNS** } "

" { { <Subset of SQL Data Types> [] | **REFERENCES** { **BYTE** | **TEXT** } } [**AS***parameter*] | [] { <Subset of SQL Data Types> [] | **REFERENCES** { **BYTE** | **TEXT** } } [**AS***parameter*] } "

Element	Description	Restrictions	Syntax
parameter	Name that you declare here for a returned parameter of the UDR	Must be unique among returned *parameter* names of the UDR. If any returned value of the UDR has a name, then all must have names.	Identifier on page 1231

Usage

For compatibility with earlier Informix® releases, you can create SPL functions with the CREATE PROCEDURE statement. (That is, you can include a Return clause in CREATE PROCEDURE statements.) Use CREATE FUNCTION, however, to create new SPL routines that return one or more values.

After the Return clause has indicated what data types are to be returned, you can use the RETURN statement of SPL at any point in the statement block to return SPL variables that correspond to the values in the Return clause.

Limits on Returned Values

An SPL function can specify more than one data type in the Return clause.

An external function (a function written in the C or the Java™ language) can specify only one data type in the Return clause, but an external function can return more than one row of data if it is an iterator function. For more information, see ITERATOR on page 1277.

Subset of SQL Data Types

The built-in SQL data types that a user-defined function (UDF) can return are language-dependent.

For more information, see the table that follows. See also Data Type on page 985.

UDFs written in a given language can return values of any built-in data type except the types that are marked with an X in the following table.

Data Type	C	Java™	SPL
BIGSERIAL	X	X	X
BLOB	X		
CLOB	X		
BYTE	X	X	
TEXT	X	X	
BSON	?	X	?
JSON	?	X	?
COLLECTION		X	

Data Type	C	Java™	SPL
LIST		X	
MULTISET		X	
ROW		X	
SET		X	
SERIAL	X	X	X
SERIAL8	X	X	X

In Informix®, if you use a complex data type in the Return clause, the calling user-defined routine must define variables of the appropriate complex types to hold the values that the C or SPL user-defined function returns.

User-defined functions can return a value of opaque or distinct data types that are defined in the database.

The default precision of a DECIMAL value that an SPL function returns is 16 digits. For a function to return a DECIMAL with a different number of significant digits, you must specify the returned precision explicitly in the *data type* specification of the Return clause.

Using the REFERENCES Clause to Point to a Simple Large Object

A user-defined function cannot return a BYTE or TEXT value (collectively called *simple large objects*) directly. A user-defined function can, however, use the REFERENCES keyword to return a descriptor that contains a pointer to a BYTE or TEXT object. The following example shows how to select a TEXT column within an SPL routine and then return the value:

```
CREATE FUNCTION sel_text()
   RETURNING REFERENCES text;
   DEFINE blob_var REFERENCES text;
   SELECT blob_col INTO blob_var
      FROM blob_table WHERE key_col = 10;
   RETURN blob_var;
END FUNCTION;
```

For simple large objects that are column values from the Projection list of a query, as in this example, the pointer in the returned descriptor references the **sysblobs.spacename** value from the system catalog, based on the BYTE or TEXT column definition.

For simple large objects that do not correspond to columns of permanent tables, however, the pointer references the dbspace of the database in which the UDR is defined. This is the default storage location for a BYTE or TEXT object that a UDR returns, when no location from the **sysblobs** table is known to the database server.

The DB-Access session in the following example creates two routines, udr1 and udr2, that each return the descriptor of a TEXT object:

```
CREATE DATABASE db WITH LOG;

CREATE TABLE t (c2 TEXT);
CREATE TABLE t1 (c2 TEXT);
```

```
LOAD FROM "t.unl" INSERT INTO t;

CREATE FUNCTION udr1 ( param_1
   REFERENCES TEXT DEFAULT NULL )
   RETURNING REFERENCES TEXT
      WITH (NOT VARIANT)
   DEFINE var1 REFERENCES TEXT;
   ON EXCEPTION
      RETURN param_1;
   END EXCEPTION;
   SELECT t.c2 udr1_col1
      INTO var1 FROM t;
   RETURN var1;
END FUNCTION;

CREATE PROCEDURE udr2 ( OUT param_1
   REFERENCES TEXT DEFAULT NULL )
      RETURNING INT;
   SELECT t.c2 udr1_col1
      INTO param_1 FROM t;
   RETURN 1;
END PROCEDURE;

SELECT udr1(t.c2) query_1_col1 FROM t
   INTO TEMP mytemp;

SELECT c2, slv1 FROM t1
   WHERE udr2(slv1#TEXT) > 0
   INTO TEMP mytemp;
```

In the SELECT statements that call these UDRs, the TEXT object that each query returns to the mytemp temporary table are stored in the dbspace of the db database.

Cursor and Noncursor Functions

A *cursor* function can fetch returned values one by one by iterating the generated result set of returned values. Such a function is an *implicitly iterated function*.

A function that returns only one set of values (such as one or more columns from a single row of a table) is a *noncursor* function.

The Return clause is valid in a cursor function or in a noncursor function. In the following example, the Return clause can return zero (`0`) or one value in a noncursor function. In a cursor function, however, it returns more than one row from a table, and each returned row contains zero or one value:

```
RETURNING INT;
```

In the following example, the Return clause can return zero (`0`) or two values if it occurs in a noncursor function. In a cursor function, however, it returns more than one row from a table, and each returned row contains zero or two values:

```
RETURNING INT, INT;
```

In both of the preceding examples, the receiving function or program must be written appropriately to accept the information that the function returns.

Routine modifier

A routine modifier specifies characteristics of how a user-defined routine (UDR) behaves.

Syntax

(explicit id unique_107_Connect_42_romo001) unique_107_Connect_42_romo001(explicit id unique_107_Connect_42_romo002)

unique_107_Connect_42_romo002(explicit id unique_107_Connect_42_romo003) unique_107_Connect_42_romo003(explicit id

unique_107_Connect_42_romo004) unique_107_Connect_42_romo004

"{ <Adding or Modifying a Routine Modifier> [] <Dropping a Routine Modifier> }"

Dropping a Routine Modifier

"{{{ [,] | [NOT] VARIANT } | NEGATOR } | [] { CLASS | ITERATOR | PARALLELIZABLE } | [] { HANDLESNULLS | INTERNAL | {

COSTFUNC | PERCALL_COST } | { SELFUNC | SELCONST } | STACK } }"

Element	Description	Restrictions	Syntax
parameter	Name that you declare here for a returned parameter of the UDR	Must be unique among returned parameters of UDRs. If any returned value of the UDR has a name, then all must have names.	Identifier on page 1231

Usage

If you drop an existing modifier in an ALTER FUNCTION, ALTER PROCEDURE, or ALTER ROUTINE statement, the database server sets the value of the modifier to the default value, if a default exists.

Some modifiers are available only with user-defined functions. For information about whether a specific routine modifier applies only to user-defined functions (that is, if it does not apply to user-defined procedures), see the description of the modifier in the sections that follow. In these sections, as elsewhere in this document, *external* refers to UDRs written in the C or Java™ languages. Features valid for only one language are so designated in the previous diagrams.

Except for VARIANT and NOT VARIANT modifiers, none of the options in this segment are valid for SPL routines.

Example

The following statement includes an external routine reference for a Java™ language UDR. You must first register *demo_jar* using the procedure **install_jar**(*<absolute path><jar file name>,<internal registered name>*).

```
CREATE FUNCTION delete_order(int) RETURNING int
   WITH (NOT VARIANT)
   EXTERNAL NAME 'informix.demo_jar:delete_order.delete_order()'
   LANGUAGE JAVA;
```

Modifier Descriptions

The following sections describe the modifiers that you can use to help the database server optimally execute a UDR.

CLASS

Use the CLASS modifier to specify the name of a virtual-processor (VP) class in which to run an external routine. A user-defined VP class must be defined before the UDR can be invoked.

You can execute C UDRs in the following types of VP classes:

- The CPU virtual-processor class (CPU VP)
- A user-defined virtual-processor class.

If you omit the CLASS modifier to specify a VP class for a UDR written in C, the UDR runs in the CPU VP. User-defined VP classes protect the database server from ill-behaved C UDRs. An ill-behaved C UDR has at least one of the following characteristics:

- It runs in the CPU VP for a long time without yielding.
- It is not thread safe.
- It calls an unsafe operating-system routine.

A well-behaved C UDR has none of these characteristics. Execute only well-behaved C UDRs in the CPU VP.

Warning: Execution of an ill-behaved C UDR in the CPU VP can cause serious interference with the operation of the database server, and the UDR might not produce correct results. For a discussion of ill-behaved UDRs, see the *HCL® Informix® DataBlade® API Programmer's Guide*.

By default, a UDR written in Java™ runs in a Java™ virtual processor class (JVP). Therefore, the CLASS modifier is optional for a UDR written in Java™. However, use the CLASS modifier when you register a UDR written in Java™ to improve readability of your SQL statements.

COSTFUNC (C)

Use the COSTFUNC modifier to specify the cost of a UDR. The *cost* of the UDR is an estimate of the time required to execute it.

Occasionally, the cost of a UDR depends on its inputs. In that case, you can use a user-defined function to calculate a cost that depends on input values.

To execute *cost_func*, you must have Execute privilege on it and on the UDR.

HANDLESNULLS

Use the HANDLESNULLS modifier to specify that a C UDR can handle NULL values that are passed to it as arguments. If you do not specify HANDLESNULLS for a C language UDR, and if you pass to it an argument that has a NULL value, the UDR does not execute and returns a NULL value.

By default, a C language UDR does *not* handle NULL values.

The HANDLESNULLS modifier is not available for SPL routines because SPL routines handle NULL values by default.

INTERNAL

Use the INTERNAL modifier with an external routine to specify that an SQL or SPL statement cannot call the external routine. An external routine that is specified as INTERNAL is not considered during routine resolution. Use the INTERNAL modifier for external routines that define access methods, language managers, and so on.

By default, an external routine is *not* internal; that is, an SQL or SPL statement can call the routine.

ITERATOR

Use the ITERATOR modifier with external functions to specify that the function is an *iterator function*. An iterator function is a function that returns a single element per function call to return a set of data; that is, it is called with an initial call and zero or more subsequent calls until the set is complete.

By default, an external C or Java™ language function is *not* an iterator function.

An SPL iterator function requires the RETURN WITH RESUME statement, rather than the ITERATOR modifier.

In ESQL/C, an iterator function requires a cursor. The cursor allows the client application to retrieve the values one at a time with the FETCH statement.

For more information on how to write iterator functions, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide* and the *HCL® Informix® DataBlade® API Programmer's Guide*.

For information about using an iterator function with a virtual table interface in the FROM clause of a query, see Iterator Functions on page 720.

NEGATOR

Use the NEGATOR modifier with UDRs that return Boolean values.

The NEGATOR modifier designates another user-defined function, called a *negator function*, as a companion to the current function. A negator function takes the same arguments as its companion function, in the same order, but returns the Boolean complement.

That is, if a function returns `TRUE` for a given set of arguments, its negator function returns `FALSE` when passed the same arguments, in the same order. For example, the following functions are negator functions:

```
equal(a,b)
notequal(a,b)
```

Both functions take the same arguments, in the same order, but return complementary Boolean values. When it is more efficient to do so, the query optimizer can use the negator function instead of the function that you specify.

To invoke a user-defined function that has a negator function, you must have the Execute privilege on both functions. In addition, the function must have the same owner as its negator function.

PARALLELIZABLE

Use the PARALLELIZABLE modifier to indicate that an external routine can be executed in parallel in the context of a parallelizable data query (PDQ).

By default, an external routine is non-parallelizable; that is, it executes in sequence.

If your UDR has a complex or smart large object data type as either a parameter or a returned value, you cannot use the PARALLELIZABLE modifier.

If you specify the PARALLELIZABLE modifier for an external routine that cannot be parallelizable, the database server returns a runtime error.

A C language UDR that calls only PDQ thread-safe functions is parallelizable. These categories of functions are PDQ thread safe:

- Data handling

  An exception in this category is that collection manipulation functions (**mi_collection_***) are not PDQ thread safe.
- Session, thread, and transaction management
- Function execution
- Memory management
- Exception handling
- Callbacks
- Miscellaneous

For details of the functions that are included in each category, see the *IBM® Informix® DataBlade® API Function Reference*.

If your C language UDR calls a function that is not included in one of these categories, it is not PDQ thread safe and is therefore not parallelizable.

To parallelize Java™ language UDR calls, the database server must have multiple instances of JVPs. UDRs written in the Java™ language and that open a JDBC connection are not parallelizable.

PERCALL_COST (C)

Use the PERCALL_COST modifier to specify the approximate CPU usage cost that a UDR incurs each time it executes.

The optimizer uses the cost you specify to determine the order in which to evaluate SQL predicates in the UDR for best performance. For example, the following query has two predicates joined by a logical AND:

```
SELECT * FROM tab1 WHERE func1() = 10 AND func2() = 'abc';
```

In this example, if one predicate returns `FALSE`, the optimizer need not evaluate the other predicate.

The optimizer uses the specified cost to order the predicates so that the least expensive predicate is evaluated first. The CPU usage cost must be an integer between `1` and 2^{31}`-1`, with `1` the lowest cost and 2^{31}`-1` the most expensive.

To calculate an approximate cost per call, add the following two figures:

- The number of lines of code executed each time the UDR is called
- The number of predicates that require an I/O access

The default cost per execution is `0`. When you drop the PERCALL_COST modifier, the cost per execution returns to `0`.

SELCONST (C)

Use the SELCONST modifier to specify the selectivity of a UDR. The selectivity of the UDR is an estimate of the fraction of the rows that the query will select.

The value of selectivity constant, **selconst**, is a floating-point number between 0 and 1 that represents the fraction of the rows for which you expect the UDR to return TRUE.

SELFUNC (C)

Use the SELFUNC modifier with a C UDR to name a companion user-defined function, called a selectivity function, to the current UDR. The selectivity function provides selectivity information about the current UDR to the optimizer.

The selectivity of a UDR is an estimate of the fraction of the rows that the query will select. That is, it is an estimate of the number of times the UDR will execute.

To execute *sel_func*, you must have Execute privilege on it and on the UDR.

Concept of Selectivity

Selectivity is an attribute of queries that performs a search based on an equality condition. The selectivity of the query depends inversely on the proportion of qualifying rows. The smaller the proportion of qualifying rows among all the rows in FROM clause table objects, the more selective is the query.

For example, the following query has a search condition based on the **customer_num** column in the **customer** table:

```
SELECT * FROM customer WHERE customer_num = 102;
```

Because each row in the table has a different customer number, this query is highly selective. In contrast, the following query has low selectivity:

```
SELECT * FROM customer WHERE state = 'CA';
```

Because most of the rows in the **customer** table are for customers in California, more than half of the rows in the table would be returned.

Restrictions on the SELFUNC Modifier

The selectivity function that you specify with the SELFUNC modifier has specific requirements.

The selectivity function that you specify must satisfy the following criteria:

- It must take the same number of arguments as the current UDR.
- The data type of each argument must be SELFUNCARGS.
- It must return a value of type FLOAT between `0` and `1`, which represents the percentage of selectivity of the function. (`1` is highly selective; `0` is not at all selective.)
- It can be written in any language that the database server supports.

A user who invokes the UDR must have the Execute privilege both on that UDR and on the selectivity function that the SELFUNC modifier specifies.

Both the UDR and the selectivity function must have the same owner.

For information on how to use the **mi_funcarg*** functions to extract information about the arguments of a selectivity function, see the *HCL® Informix® DataBlade® API Programmer's Guide*.

STACK (C)

Use the STACK modifier with a C UDR to override the default stack size that the STACKSIZE configuration parameter specifies.

The STACK modifier specifies the size (in bytes) of the thread stack, which a user thread that executes the UDR uses to hold information such as routine arguments and returned values from functions.

A UDR needs to have enough stack space for all its local variables. For a particular UDR, you might need to specify a stack size larger than the default size to prevent stack overflow.

When a UDR that includes the STACK modifier executes, the database server allocates a thread-stack size of the specified number of bytes. Once the UDR completes execution, subsequent UDRs execute in threads with a stack size that the STACKSIZE configuration parameter specifies (unless any of these subsequent UDRs have also specified the STACK modifier).

For more information about the thread stack, see your *HCL® Informix® Administrator's Guide* and the *IBM® Informix® DataBlade® API Function Reference*.

VARIANT and NOT VARIANT

Use the VARIANT and NOT VARIANT modifiers with C user-defined functions and SPL functions. A function is *variant* if it returns different results when it is invoked with the same arguments or if it modifies a database or variable state. For example, a function that returns the current date or time is a variant function.

By default, user-defined functions are variant. If you specify NOT VARIANT when you create or modify a user-defined function, the function cannot contain any SQL statements.

If the user-defined function is nonvariant, the database server might cache the returned values of expensive functions. You can create functional indexes only on nonvariant functions. For more information on functional indexes, see CREATE INDEX statement on page 237.

In ESQL/C, you can specify VARIANT or NOT VARIANT in this clause or in the EXTERNAL Routine Reference. For more information, see External Routine Reference on page 1229. If you specify the modifier in both places, however, you must use the same modifier in both clauses.

Routine Parameter List

Use the appropriate part of the Routine Parameter List segment whenever you see a reference to a Routine Parameter List in a syntax diagram.

Syntax

(explicit id unique_307_Connect_42_ropal002) unique_307_Connect_42_ropal002(explicit id unique_307_Connect_42_ropal003)

unique_307_Connect_42_ropal003

Routine Parameter List

" [{ **IN** | **OUT** | **INOUT** }] <Parameter> "

Parameter

" { *parameter* | [] } "

" { { <Subset of SQL Data Type> [] | **LIKE** *table*.*column* } [**DEFAULT***value*] | **REFERENCES** { **BYTE** | **TEXT** } [**DEFAULT** **NULL**] } "

Element	Description	Restrictions	Syntax
column	Name of a column whose data type is declared for *parameter*	Must exist in the specified table	Database Object Name on page 1225
parameter	Name of a parameter of the UDR	Name is required for SPL routines	Identifier on page 1231
table	Table that contains *column*	The table must exist in the database	Identifier on page 1231

Element	Description	Restrictions	Syntax
value	Default used if UDR is called with no value for *parameter*	Must be a literal, of the same data type as *parameter*. For opaque types, an input function must be defined	Literal Number on page 1198

Usage

A *parameter* is a formal argument in the declaration of a UDR. (When you subsequently invoke a UDR that has parameters, you must substitute a specific argument value for the parameter, unless the parameter has a default value.)

The name of the parameter is optional for external routines of HCL Informix®.

When you create a UDR, you declare a *name* and *data type* for each parameter. You can specify the data type directly, or use the LIKE or REFERENCES clause to specify the data type. You can optionally specify a default value.

You can define any number of SPL routine parameters, but the total length of all parameters passed to an SPL routine must be less than 2 gigabytes.

No more than nine arguments to a UDR written in the Java™ language can be DECIMAL data types of SQL that the UDR declares as BigDecimal data types of the Java™ language.

Any C language UDR that returns an opaque data type must specify **opaque_type** in the `var binary` declaration of the C host variable.

Subset of SQL Data Types

Serial and large-object data types are not valid as parameters. A UDR can declare a parameter of any other data type defined in the database, including any built-in data types except BIGSERIAL, BLOB, BYTE, CLOB, SERIAL, SERIAL8, or TEXT.

On Informix®, a parameter can also be a complex data type or a UDT, but complex data types are not valid for parameters of external UDRs written in the Java™ language.

For information about the data types of Informix® that are valid as parameters or return values of routines that access tables or views outside the local database, see #unique_1760.

Using the LIKE Clause

Use the LIKE clause to specify that the data type of a parameter is the same as a column defined in the database. If the ALTER TABLE statement changes the data type of the column, the data type of the parameter also changes.

In Informix®, if you use the LIKE clause to declare any parameter, you cannot overload the UDR. For example, suppose you create the following user-defined procedure:

```
CREATE PROCEDURE cost (a LIKE tableX.colY, b INT)
. . .
END PROCEDURE;
```

You cannot create another procedure named **cost( )** in the same Informix® database with two arguments. You can, however, create a procedure named **cost( )** with a number of arguments other than two. (Another way to circumvent this restriction on the LIKE clause is through user-defined data types.)

Using the REFERENCES Clause

Use the REFERENCES clause to specify that a parameter contains BYTE or TEXT data. The REFERENCES keyword allows you to use a pointer to a BYTE or TEXT object as a parameter. If you use the DEFAULT NULL option in the REFERENCES clause, and you call the UDR without a parameter, a NULL value is used as the default value.

Using the DEFAULT Clause

Use the DEFAULT keyword followed by an expression to specify a default value for a parameter. If you provide a default value for a parameter, and the UDR is called with fewer arguments than were defined for that UDR, the default value is used. If you do not provide a default value for a parameter, and the UDR is called with fewer arguments than were defined for that UDR, the calling application receives an error.

The following example shows a CREATE FUNCTION statement that specifies a default value for a parameter. This function finds the square of the *i* parameter. If the function is called without specifying the argument for the *i* parameter, the database server uses the default value `0` for the *i* parameter.

```
CREATE FUNCTION square_w_default
   (i INT DEFAULT 0) {Specifies default value of i}
   RETURNING INT; {Specifies return of INT value}
   DEFINE j INT; {Defines routine variable j}
   LET j = i * i; {Finds square of i and assigns it to j}
   RETURN j; {Returns value of j to calling module}
END FUNCTION;
```

Warning: When you specify a date value as the default value for a parameter, make sure to specify 4 digits instead of 2 digits for the year. When you specify a 2-digit year, the **DBCENTURY** environment variable setting can affect how the database server interprets the date value, so the UDR might not use the default value that you intended. For more information, see the *HCL® Informix® Guide to SQL: Reference*.

Specifying OUT Parameters for User-Defined Routines

When you register a user-defined routine of Informix®, you can use the OUT keyword to specify that any parameter in the list is an OUT parameter. Each OUT parameter corresponds to a value the routine returns indirectly, through a pointer. The value that the routine returns through the pointer is an extra value, in addition to any values that it returns explicitly.

After you have registered a user-defined function that has one or more OUT parameters, you can use the function with a statement-local variable (SLV) in a SELECT statement. (For information about statement-local variables, see Statement-Local Variable Expressions on page 1153.)

If you specify any OUT parameters, and you use Informix®-style parameters, the arguments are passed to the OUT parameters by reference. The OUT parameters are not significant in determining the routine signature.

For example, the following declaration of a C user-defined function allows you to return an extra value through the **y** parameter:

```
int my_func( int x, int *y );
```

Register the C function with a CREATE FUNCTION statement similar to this:

```
CREATE FUNCTION my_func( x INT, OUT y INT )
   RETURNING INT
   EXTERNAL NAME "/usr/lib/local_site.so"
   LANGUAGE C
END FUNCTION;
```

In the next example, this Java™ method returns an extra value by passing an array:

```
public static String allVarchar(String arg1, String[] arg2)
throws SQLException
{
arg2[0] = arg1;
return arg1;
}
```

To register this as a UDF, use a statement similar to the following example:

```
CREATE FUNCTION all_varchar(VARCHAR(10), OUT VARCHAR(7))
   RETURNING VARCHAR(7)
   WITH (class = "jvp")
   EXTERNAL NAME 'informix.testclasses.jlm.Param.allVarchar(java.lang.String,
   java.lang.String[ ])'
   LANGUAGE JAVA;
```

Specifying INOUT Parameters for a User-Defined Routine

UDRs that are written in the SPL, C, or Java™ languages can also support INOUT parameters. When the UDR is invoked, a value for each INOUT parameter is passed by reference as an argument to the UDR.

When the UDR completes execution, it can return a modified value for the INOUT parameter to the calling context. The INOUT parameter can be of any data type that Informix® supports, including user-defined and complex data types, with the following exceptions:

- Serial types (BIGSERIAL, SERIAL, and SERIAL8)
- Simple large object types (BYTE and TEXT).

In the following example, the CREATE PROCEDURE statement registers a C routine that has a single INOUT parameter:

```
 CREATE PROCEDURE CALC ( INOUT param1 float )
    EXTERNAL NAME "$INFORMIXDIR/etc/myudr.so(calc)"
        LANGUAGE C;
```

An SPL routine can invoke other UDRs that have OUT or INOUT parameters, if those UDRs are written in the SPL or C language. An SPL routine cannot, however, invoke a Java™ UDR whose arguments include OUT or INOUT parameters.

Support for invoking UDRs that have named or unnamed ROW arguments from an SPL routine has the following dependencies on the parameter type of the ROW argument, and on the programming language of the invoked UDR:

- SPL routines can invoke C UDRs that have ROW arguments that are IN parameters, but cannot invoke C UDRs that have ROW arguments that are OUT or INOUT parameters.
- SPL routines can invoke SPL UDRs that have ROW arguments of any parameter type, including IN, OUT, and INOUT.

You can assign INOUT parameters to statement-local variables (SLVs), which the section Statement-Local Variable Expressions on page 1153 describes.

Shared-Object Filename

Use a shared-object filename to specify a pathname to an executable object file when you register or alter an external routine.

Syntax

(explicit id unique_108_Connect_42_sofi001) unique_108_Connect_42_sofi001(explicit id unique_108_Connect_42_sofi002)

unique_108_Connect_42_sofi002(explicit id unique_108_Connect_42_sofi003) unique_108_Connect_42_sofi003(explicit id unique_108_Connect_42_sofi004)

unique_108_Connect_42_sofi004

Shared-Object File



Usage

If the IFX_EXTEND_ROLE configuration parameter is set to `1` or to ON, only users to whom the DBSA has granted the built-in EXTEND role are authorized to use this segment. (Whether or not IFX_EXTEND_ROLE is enabled, you must hold the Resource privilege or the DBA privilege on the database, and you must also hold the Usage privilege on the external programming language in which the UDR is written, before you can create, drop, or alter an external UDR.)

The Database Server Administrator should include in the DB_LIBRARY_PATH configuration parameter settings every file system where the security policy authorizes modules and UDRs to reside. Unless DB_LIBRARY_PATH is absent or has no setting, the database server cannot access a file that this segment specifies unless its pathname begins with a string that exactly matches one of the values of DB_LIBRARY_PATH.

For example, if "**$INFORMIXDIR/extend**" is one of the DB_LIBRARY_PATH values on a Linux™ system, then shared-object files can have pathnames within the **$INFORMIXDIR/extend** file system or its subdirectories. This directory is also the file system where built-in modules reside.

The syntax by which you specify a shared-object filename depends on whether the external routine is written in the C language or in the Java™ language. Sections that follow describe each of these external languages.

For more information about the context in which a shared-object filename appears within EXTERNAL NAME clause of the ALTER FUNCTION, ALTER PROCEDURE, ALTER ROUTINE, CREATE FUNCTION, and CREATE PROCEDURE statements, see the related reference, External Routine Reference.

C Shared-Object File

To specify the location of a C shared-object file, you can specify the path to the dynamically-loaded executable file within a quoted pathname or as a variable.

The C shared-object filename is specified by the following syntax:

C Shared-Object File

" { *quote* [{ $*environment_var* | / | . }] *pathname* [*(symbol)*] *quote* | $*variable* } "

Element	Description	Restrictions	Syntax
environment_var	Platform-independent indicator	Must begin with a dollar sign ($)	Identifier on page 1231
pathname	Pathname to the file	See notes that follow this table	Must conform to operating system conventions
quote	Either single (') or double (") quotation mark symbol	Opening and closing quotation mark symbols must match	Literal symbol (either ' or ")
symbol	Entry point to the file	Must be enclosed in parentheses	Must conform to operating system conventions
variable	Platform-independent indicator	Must begin with a dollar sign ($)	Must conform to C language conventions

The following rules affect *pathname* and filename specifications in C:

- A filename (with no *pathname*) can specify an internal function.
- You can omit the period (.) symbol if *pathname* is relative to the current directory when the CREATE or ALTER statement is run.
- On UNIX™, an absolute pathname must begin with a slash (/) symbol, and each directory name must end with a slash (/) symbol.
- On Windows™, an absolute pathname must begin with a backslash (\ ) symbol, and each directory name must end with a backslash (\ ) symbol.
- The filename at the end of *pathname* must have the **.so** file extension and must refer to an executable file in a shared object library.
- Use a *symbol* only if the entry point to the dynamically loadable executable object file has a different name from the UDR that you are registering with CREATE FUNCTION or CREATE PROCEDURE.

- If you specify a *variable*, it must contain the full pathname to the executable file.
- You can include white-space characters, such as blank spaces or tab characters, within a quoted pathname.

Java™ Shared-Object File

To specify the name of a Java™ shared-object file, specify the name of the static Java™ method to which the UDR corresponds and the location of the Java™ binary that defines the method.

(explicit id unique_1765_Connect_42_jsof003) unique_1765_Connect_42_jsof003

Java Shared-Object File

" *quote* "

" <Jar Name>[] "

" : "

" [*package_id* .] "

" *class_id* . *method_id* "

" [([*java_type*]) **RETURNS** *java_type*] "

" *quote* "

Element	Description	Restrictions	Syntax
class_id	Java™ class whose method implements the UDR	Class must exist in the **.jar** file that Jar Name identifies	Must conform to rules for Java™ identifiers
java_type	Java™ data type for a parameter in the Java-method signature	Must be defined in a JDBC class or by an SQL-to-Java mapping	Must conform to rules for Java™ identifiers
method_id	Name of the Java™ method that implements the UDR	Must exist in the Java™ class that *java_class_name* specifies	Must conform to rules for Java™ identifiers
package_id	Name of package that contains the Java™ class	Must exist	Must conform to rules for Java™ identifiers
quote	Single (') or double (") quotation mark delimiters	Opening and closing quotation marks must match	Literal symbol (' or ") entered at the keyboard

Before you can create a UDR written in the Java™ language, you must assign a jar identifier to the external jar file with the **sqlj.install_jar** procedure. (For more information, see sqlj.install_jar.) You can include the Java™ signature of the method that implements the UDR in the shared-object filename.

- If you do *not* specify the Java™ signature, the routine manager determines the *implicit* Java™ signature from the SQL signature in the CREATE FUNCTION or CREATE PROCEDURE statement.

  It maps SQL data types to the corresponding Java™ data types with the JDBC and SQL-to-Java mappings. For information on mapping user-defined data types to Java™ data types, see sqlj.setUDTextName.
- If you do specify the Java™ signature, the routine manager uses this *explicit* Java™ signature as the name of the Java™ method to use.

For example, if the Java™ method **explosiveReaction( )** implements the Java™ UDR **sql_explosive_reaction( )** as discussed in sqlj.install_jar, its shared-object filename could be:

```
course_jar:Chemistry.explosiveReaction
```

The preceding shared-object filename provides an implicit Java™ signature. The following shared-object filename is the equivalent with an explicit Java™ signature:

```
course_jar:Chemistry.explosiveReaction(int)
```

Specific Name

Use a specific name to declare an identifier for a UDR that is unique in the database or name space. Use the Specific Name segment whenever you see a reference to a specific name in a syntax diagram.

Syntax

Specific Name

" [*owner* .] "

" *specific_id* "

Element	Description	Restrictions	Syntax
owner	Owner of the UDR	No more than 32 bytes. Must be same as *owner* of function or procedure name of this UDR. See also Restrictions on the Owner Name on page 1289.	Owner name on page 1262
specific_id	Unique name of the UDR	Must be no more than 128 bytes long. See also Restrictions on the Specific Name on page 1289.	Identifier on page 1231

Usage

A *specific name* is a unique identifier that the CREATE PROCEDURE or CREATE FUNCTION statement declares as an alternative name for a UDR.

Because you can overload routines, a database can have more than one UDR with the same name and different parameter lists. You can assign a UDR a specific name that uniquely identifies the specific UDR.

If you declare a specific name when you create the UDR, you can later use that name when you alter, drop, grant, or revoke privileges, or update statistics on that UDR. Otherwise, you need to include the parameter data types with the UDR name, if the name alone does not uniquely identify the UDR.

Restrictions on the Owner Name

When you declare a specific name, the *owner* must be the same *authorization identifier* that qualifies the function name or procedure name of the UDR that you create. That is, whether or not you specify the owner name to qualify either the UDR name or the specific name or both, the names of the *owner* must match.

When you specify no owner name in the DDL statement that creates a UDR, Informix® uses the login name of the user who creates the UDR. Therefore, if you specify the owner name in one location and not the other, the owner name that you specify must match your user ID.

Restrictions on the Specific Name

In a database that is not ANSI-compliant, *specific_id* must be unique among routine names within the database. Two UDRs cannot have the same *specific_id*, even if they have different owners.

In an ANSI-compliant database, the combination *owner.specific_id* must be unique. That is, the specific name must be unique among UDRs that have the same owner.

Statement Block

Use a statement block to specify SPL and SQL operations to take place when an SPL statement that includes this segment is executed.

Syntax

(explicit id unique_310_Connect_42_sbl002) unique_310_Connect_42_sbl002(explicit id unique_310_Connect_42_sbl003)

unique_310_Connect_42_sbl003(explicit id unique_310_Connect_42_sbl004) unique_310_Connect_42_sbl004(explicit id unique_310_Connect_42_sbl005)

unique_310_Connect_42_sbl005(explicit id unique_310_Connect_42_sbl006) unique_310_Connect_42_sbl006(explicit id unique_310_Connect_42_sbl007)

unique_310_Connect_42_sbl007

Statement Block

" [<DEFINE Statement>[]] "

" [<ON EXCEPTION Statement>[]] "

" [{| <EXECUTE FUNCTION Statement>[] | <EXECUTE PROCEDURE Statement> [] | <Subset of SPL Statements>[] | <Subset of SQL Statements>[] ; | **BEGIN** <Statement Block>**END** }] "

Usage

SPL and SQL statements can appear in a *statement block*, a set of zero or more statements that can define the scope of a variable or of the ON EXCEPTION statement. If a statement block is empty, no operation takes place when control of execution within the SPL routine passes to the empty SPL statement block.

Subset of SPL Statements Valid in the Statement Block

The diagram for the Statement Block on page 1289 refers to this section. You can use any of the following SPL statements in the statement block:

- <<*Label* >>
- CALL
- CASE
- CONTINUE
- EXIT
- FOR
- FOREACH
- GOTO
- IF
- LET
- LOOP
- RAISE EXCEPTION
- RETURN
- SYSTEM
- TRACE
- WHILE

GOTO and << *Label* >>, however, are not valid in ON EXCEPTION statement blocks.

SQL Statements Valid in SPL Statement Blocks

The diagram for the Statement Block on page 1289 refers to this section. Most SQL statements are valid in SPL statement blocks, except for the statements that are listed below. The following SQL statements are *not* valid in an SPL statement block:

- CLOSE DATABASE
- CONNECT
- CREATE DATABASE
- CREATE FUNCTION
- CREATE FUNCTION FROM
- CREATE PROCEDURE
- CREATE PROCEDURE FROM
- CREATE ROUTINE FROM
- DATABASE
- DISCONNECT
- DROP DATABASE
- EXECUTE
- FLUSH
- INFO
- LOAD
- OUTPUT
- PUT
- RENAME DATABASE
- SET AUTOFREE
- SET CONNECTION
- UNLOAD
- UPDATE STATISTICS

For example, you cannot close the current database or connect to a new database within an SPL routine. Similarly, you cannot drop the current SPL routine within the same routine. You can, however, drop another SPL routine.

Only two forms of the SELECT statement are valid in queries within SPL routines:

- You can use the INTO TEMP clause to put the result of the SELECT statement into a temporary table.
- You can use the SELECT ... INTO form of the SELECT statement to put the resulting values into SPL variables.

When you include the ORDER BY clause in the SELECT ... INTO TEMP or the SELECT ... INTO *variable* statement, you imply that the query returns more than one row. The database server issues an error if you specify the ORDER BY clause without a FOREACH loop to process the returned rows individually within the SPL routine.

If an SPL routine is called as part of a data-manipulation language (DML) statement, additional restrictions exist. For more information, see Restrictions on SPL Routines in Data-Manipulation Statements on page 1293.

Nested Statement Blocks

You can use the BEGIN and END keywords to delimit a statement block that is nested within another statement block.

Scope of Reference of SPL Variables and Exception Handlers

The BEGIN and END keywords can limit the scope of SPL variables and exception handlers. Declarations of variables and definitions of exception handlers inside a BEGIN and END statement block are local to that statement block and are not visible from outside the statement block. The following code uses a BEGIN and END statement block to delimit the scope of reference of variables:

```
CREATE DATABASE demo;
CREATE TABLE tracker (
   who_submitted CHAR(80), -- Show what code was running.
   value INT,              -- Show value of the variable.
   sequential_order SERIAL -- Show order of statement execution.
    );
CREATE PROCEDURE demo_local_var()
DEFINE var1, var2 INT;
   LET var1 = 1;
   LET var2 = 2;
   INSERT INTO tracker (who_submitted, value)
    VALUES ('var1 param before sub-block', var1);
BEGIN
   DEFINE var1 INT;   -- same name as global parameter.
   LET var1 = var2;
   INSERT INTO tracker (who_submitted, value)
   VALUES ('var1 var defined inside the "IF/BEGIN".', var1);
END
INSERT INTO tracker (who_submitted, value)
   VALUES ('var1 param after sub-block (unchanged!)', var1);
END PROCEDURE;
EXECUTE PROCEDURE demo_local_var();
SELECT sequential_order, who_submitted, value FROM tracker
 ORDER BY sequential_order;
```

This example declares three variables, two of which are named **var1**. (Name conflicts are created here to illustrate which variables are visible. Using the same name for different variables is generally not recommended, because conflicting names of variables can make your code more difficult to read and to maintain.)

Because of the statement block, only one **var1** variable is in scope at a time.

The **var1** variable that is declared inside the statement block is the only **var1** variable that can be referenced from within the statement block.

The **var1** variable that is declared outside the statement block is not visible within the statement block. Because it is out of scope, it is unaffected by the change in value to the **var1** variable that takes place inside the statement block. After all the statements run, the outer **var1** still has a value of `1`.

The **var2** variable is visible within the statement block because it was not superseded by a name conflict with a block-specific variable.

Restrictions on SPL Routines in Data-Manipulation Statements

If you call the SPL routine in a SQL statement that is not a data-manipulation language (DML) statement (namely EXECUTE FUNCTION or EXECUTE PROCEDURE), the SPL routine can execute any statement that is not listed in the section SQL Statements Valid in SPL Statement Blocks on page 1291.

If you call the SPL routine as part of a DML statement (namely, an INSERT, UPDATE, DELETE, MERGE, or SELECT statement), the routine cannot execute any of the following SQL statements:

- ALTER ACCESS_METHOD
- ALTER FRAGMENT
- ALTER INDEX
- ALTER SEQUENCE
- ALTER TABLE
- BEGIN WORK
- COMMIT WORK
- CREATE ACCESS_METHOD
- CREATE AGGREGATE
- CREATE DISTINCT TYPE
- CREATE OPAQUE TYPE
- CREATE OPCLASS
- CREATE ROLE
- CREATE ROW TYPE
- CREATE SEQUENCE
- CREATE TRIGGER
- DELETE
- DROP ACCESS_METHOD
- DROP AGGREGATE
- DROP INDEX
- DROP OPCLASS
- DROP ROLE
- DROP ROW TYPE
- DROP SEQUENCE
- DROP SYNONYM
- DROP TABLE
- DROP TRIGGER
- DROP TYPE
- DROP VIEW
- INSERT
- MERGE

- RENAME COLUMN
- RENAME DATABASE
- RENAME SEQUENCE
- RENAME TABLE
- ROLLBACK WORK
- SET CONSTRAINTS
- TRUNCATE
- UPDATE

HCL Informix® issues error -675 if an SPL routine whose calling context is a DML statement attempts to execute any of the SQL statements listed above.

These restrictions do not apply to an SPL routine that is invoked by a trigger, because in this case the SPL routine is not called by the DML statement, and therefore can include any SQL statement, such as UPDATE, INSERT and DELETE, that is not listed among the SQL Statements Valid in SPL Statement Blocks on page 1291.

Transactions in SPL Routines

In a database that is not ANSI-compliant, you can use the BEGIN WORK and COMMIT WORK statements in an SPL statement block to start a transaction, to finish a transaction, or start and finish a transaction in the same SPL routine. If you start a transaction in a routine that is executed remotely, you must finish the transaction before the routine exits.

As previously noted, however, the ROLLBACK WORK statement is not valid in an SPL statement block.

Support for roles and user identity

You can use roles with SPL routines. You can execute role-related SQL statements (CREATE ROLE, DROP ROLE, GRANT, REVOKE, and SET ROLE) and a user who holds the SETSESSIONAUTH privilege can issue SET SESSION AUTHORIZATION statements in an SPL routine. Within an SPL routine, you can also use the GRANT statement

- to grant discretionary access privileges to roles,
- or grant label-based access credentials (LBAC) to roles,
- or grant other roles to roles.

An SPL routine can also use the REVOKE statement to cancel the access privileges, the LBAC credentials, or the roles that a role holds.

Access privileges, roles, and LBAC credentials that a user has acquired in an SPL routine by enabling a role or by a SET SESSION AUTHORIZATION statement are not automatically relinquished after the SPL routine that granted the privilege, role, or LBAC credential completes execution. What was granted persists until a subsequent REVOKE operation cancels the effect of the GRANT operation.

For further information about roles, see the CREATE ROLE statement on page 282, DROP ROLE statement on page 481, GRANT statement on page 537, REVOKE statement on page 654, and SET ROLE statement on page 829 in Chapter 2.

Appendixes

Keywords of SQL for HCL Informix®

This appendix lists the keywords in the HCL Informix® implementation of SQL for Informix®.

The ISO standard SQL language has many keywords. Some are designated as *reserved words* and others as *non-reserved words*. In ISO SQL, reserved words cannot be used as identifiers for database objects, such as tables, columns, and so on. To use a reserved word as a name in a valid SQL statement requires a delimited identifier (Delimited Identifiers on page 1233) that you enclose between double (`" "`) quotation marks.

In contrast, the dialect of SQL that HCL Informix® database servers implement has few reserved words in the sense of a character string that obeys the rules for identifiers (Identifier on page 1231) but always produces a compilation error or runtime error when used as an identifier. Your application might encounter restricted functionality, however, or unexpected results, if you define an SPL routine that has the same name as a built-in SQL function, expression, or operator.

Do not declare any of the keywords in this appendix as SQL identifiers. If you do, errors or syntactic ambiguities can occur if the identifier appears in a context where the keyword is valid. In addition, your code will be more difficult to read and to maintain. Informix® reserves the prefixes ifx_ and **sys** for built-in routines and database objects. Do not use keywords of C or C++ (or of any other programming language that you will be using in an embedded mode) in your database structures. The notations IFX_* and SYS* (where * is a wildcard character for "any string") in the alphabetized lists that follow. These indicate that those prefixes should be avoided in user-defined identifiers of database objects. (Keywords of SQL that begin with those suffixes are not listed in this appendix, whose goal is to assist HCL Informix® users in avoiding names that are used internally by the database server.)

If you receive an error message that seems unrelated to the SQL statement that caused the error, review this appendix to see whether a keyword is used as an identifier.

To avoid using a keyword as an identifier, you can qualify the identifier with an owner name or modify the identifier. For example, rather than name a database object **CURRENT**, you might name it **o_current** or **juanita.current**. For a discussion of potential problems in using keywords as identifiers, and of additional workarounds for specific keywords, see Use of Keywords as Identifiers on page 1233. See also *HCL® Informix® Guide to SQL: Tutorial* for more information about using keywords as identifiers in SQL applications.

A

AAO
ABS
ABSOLUTE
ACCELERATE
ACCESS
ACCESS_METHOD
ACCOUNT

ACOS
ACOSH
ACTIVE
ADD
ADDRESS
ADD_MONTHS
ADMIN
AFTER
AGGREGATE
ALIGNMENT
ALL
ALL_ROWS
ALLOCATE
ALTER
AND
ANSI
ANY
APPEND
AQT
ARRAY
AS
ASC
ASCII
ASIN
ASINH
ASYNC
AT
ATAN
ATAN2
ATANH
ATTACH
ATTRIBUTES
AUDIT
AUTHENTICATION
AUTHID
AUTHORIZATION
AUTHORIZED
AUTO
AUTOFREE
AUTOLOCATE
AUTO_READAHEAD
AUTO_REPREPARE

AUTO_STAT_MODE
AVG
AVOID_EXECUTE
AVOID_FACT
AVOID_FULL
AVOID_HASH
AVOID_INDEX
AVOID_INDEX_SJ
AVOID_MULTI_INDEX
AVOID_NL
AVOID_STAR_JOIN

B

BARGROUP
BASED
BEFORE
BEGIN
BETWEEN
BIGINT
BIGSERIAL
BINARY
BITAND
BITANDNOT
BITNOT
BITOR
BITXOR
BLOB
BLOBDIR
BOOLEAN
BOTH
BOUND_IMPL_PDQ
BSON
BUCKETS
BUFFERED
BUILTIN
BY
BYTE

C

CACHE
CALL
CANNOTHASH
CARDINALITY
CASCADE
CASE
CAST
CEIL
CHAR
CHAR_LENGTH
CHARACTER
CHARACTER_LENGTH
CHARINDEX
CHECK
CHR
CLASS
CLASS_ORIGIN
CLEANUP
CLIENT
CLOB
CLOBDIR
CLOSE
CLUSTER
CLUSTER_TXN_SCOPE
COBOL
CODESET
COLLATION
COLLECTION
COLUMN
COLUMNS
COMMIT
COMMITTED
COMMUTATOR
COMPONENT
COMPONENTS
COMPRESSED
CONCAT
CONCURRENT
CONNECT
CONNECTION

CONNECTION_NAME
CONNECT_BY_ISCYCLE
CONNECT_BY_ISLEAF
CONNECT_BY_ROOT
CONST
CONSTRAINT
CONSTRAINTS
CONSTRUCTOR
CONTEXT
CONTINUE
COPY
COS
COSH
COSTFUNC
COUNT
CRCOLS
CREATE
CROSS
CUME_DIST
CURRENT
CURRENT_ROLE
CURRENT_USER
CURRVAL
CURSOR
CYCLE

D

DATA
DATABASE
DATAFILES
DATASKIP
DATE
DATETIME
DAY
DBA
DBDATE
DBINFO
DBPASSWORD
DBSA
DBSERVERNAME
DBSECADM

DBSSO
DEALLOCATE
DEBUG
DEBUGMODE
DEBUG_ENV
DEC
DECIMAL
DECLARE
DECODE
DECRYPT_BINARY
DECRYPT_CHAR
DEC_T
DEFAULT
DEFAULTESCCHAR
DEFAULT_ROLE
DEFAULT_USER
DEFERRED
DEFERRED_PREPARE
DEFINE
DEGREES
DELAY
DELETE
DELETING
DELIMITED
DELIMITER
DELUXE
DENSERANK
DENSE_RANK
DESC
DESCRIBE
DESCRIPTOR
DETACH
DIAGNOSTICS
DIRECTIVES
DIRTY
DISABLE
DISABLED
DISCARD
DISCONNECT
DISK
DISTINCT
DISTRIBUTEBINARY

DISTRIBUTESREFERENCES
DISTRIBUTIONS
DOCUMENT
DOMAIN
DONOTDISTRIBUTE
DORMANT
DOUBLE
DROP
DTIME_T

E

EACH
ELIF
ELSE
ENABLE
ENABLED
ENCRYPT_AES
ENCRYPT_TDES
ENCRYPTION
END
ENUM
ENVIRONMENT
ERKEY
ERROR
ESCAPE
EXCEPT
EXCEPTION
EXCLUSIVE
EXEC
EXECUTE
EXECUTEANYWHERE
EXEMPTION
EXISTS
EXIT
EXP
EXPLAIN
EXPLICIT
EXPRESS
EXPRESSION
EXTDIRECTIVES
EXTEND

EXTENT
EXTERNAL
EXTYPEID
EXTYPELENGTH
EXTYPENAME
EXTYPEOWNERLENGTH
EXTYPEOWNERNAME

F

FACT
FALLBACK
FALSE
FAR
FETCH
FILE
FILETOBLOB
FILETOCLOB
FILLFACTOR
FILTERING
FINAL
FIRST
FIRST_ROWS
FIRST_VALUE
FIXCHAR
FIXED
FLOAT
FLOOR
FLUSH
FOLLOWING
FOR
FORCE
FORCED
FORCE_DDL_EXEC
FOREACH
FOREIGN
FORMAT
FORMAT_UNITS
FORTRAN
FOUND
FRACTION

FRAGMENT
FRAGMENTS
FREE
FROM
FULL
FUNCTION

G

G
GB
GENBSON
GENERAL
GET
GETHINT
GIB
GLOBAL
GO
GOTO
GRANT
GREATERTHAN
GREATERTHANOREQUAL
GRID
GRID_NODE_SKIP
GROUP

H

HANDLESNULLS
HASH
HAVING
HDR
HDR_TXN_SCOPE
HEX
HIGH
HINT
HOLD
HOME
HOUR

I

IDATA
IDSLBACREADARRAY
IDSLBACREADSET
IDSLBACREADTREE
IDSLBACRULES
IDSLBACWRITEARRAY
IDSLBACWRITESET
IDSLBACWRITETREE
IDSSECURITYLABEL
IF
IFX_*
ILENGTH
IMMEDIATE
IMPLICIT
IMPLICIT_PDQ
IN
INACTIVE
INCREMENT
INDEX
INDEXES
INDEX_ALL
INDEX_SJ
INDICATOR
INFORMIX
INFORMIXCONRETRY
INFORMIXCONTIME
INIT
INITCAP
INLINE
INNER
INOUT
INSENSITIVE
INSERT
INSERTING
INSTEAD
INSTR
INT
INT8
INTEG
INTEGER

INTERNAL
INTERNALLENGTH
INTERSECT
INTERVAL
INTO
INTRVL_T
IS
ISCANONICAL
ISOLATION
ITEM
ITERATOR
ITYPE

J - K

JAVA
JOIN
JSON
K
KB
KEEP
KEY
KIB

L

LABEL
LABELEQ
LABELGE
LABELGLB
LABELGT
LABELLE
LABELLT
LABELLUB
LABELTOSTRING
LAG
LANGUAGE
LAST
LAST_DAY
LAST_VALUE
LATERAL
LEAD

LEADING
LEFT
LEN
LENGTH
LESSTHAN
LESSTHANOREQUAL
LET
LEVEL
LIKE
LIMIT
LIST
LISTING
LOAD
LOCAL
LOCATOR
LOCK
LOCKS
LOCOPY
LOC_T
LOG
LOG10
LOGN
LONG
LOOP
LOTOFILE
LOW
LOWER
LPAD
LTRIM
LVARCHAR

M

M
MATCHED
MATCHES
MAX
MAXERRORS
MAXLEN
MAXVALUE
MB
MDY

MEDIAN
MEDIUM
MEMORY
MEMORY_RESIDENT
MERGE
MESSAGE_LENGTH
MESSAGE_TEXT
MIB
MIN
MINUS
MINUTE
MINVALUE
MOD
MODE
MODERATE
MODIFY
MODULE
MONEY
MONTH
MONTHS_BETWEEN
MORE
MULTISET
MULTI_INDEX

N

NAME
NCHAR
NEAR_SYNC
NEGATOR
NEW
NEXT
NEXT_DAY
NEXTVAL
NLSCASE
NO
NOCACHE
NOCYCLE
NOMAXVALUE
NOMIGRATE
NOMINVALUE
NONE

NON_RESIDENT
NON_DIM
NOORDER
NORMAL
NOT
NOTEMPLATEARG
NOTEQUAL|
NOVALIDATE
NTILE
NULL
NULLABLE
NULLIF
NULLS
NUMBER
NUMERIC
NUMROWS
NUMTODSINTERVAL
NUMTOYMINTERVAL
NVARCHAR
NVL

O

OCTET_LENGTH
OF
OFF
OLD
ON
ONLINE
ONLY
OPAQUE
OPCLASS
OPEN
OPTCOMPIND
OPTIMIZATION
OPTION
OR
ORDER
ORDERED
OUT
OUTER
OUTPUT

OVER
OVERRIDE

P

PAGE
PARALLELIZABLE
PARAMETER
PARTITION
PASCAL
PASSEDBYVALUE
PASSWORD
PDQPRIORITY
PERCALL_COST
PERCENT_RANK
PIPE
PLI
PLOAD
POLICY
POW
POWER
PRECEDING
PRECISION
PREPARE
PREVIOUS
PRIMARY
PRIOR
PRIVATE
PRIVILEGES
PROBE
PROCEDURE
PROPERTIES
PUBLIC
PUT

Q

QUARTER

R

RADIANS
RAISE

RANGE
RANK
RATIOTOREPORT
RATIO_TO_REPORT
RAW
READ
REAL
RECORDEND
REFERENCES
REFERENCING
REGISTER
REJECTFILE
RELATIVE
RELEASE
REMAINDER
RENAME
REOPTIMIZATION
REPEATABLE
REPLACE
REPLICATION
RESOLUTION
RESOURCE
RESTART
RESTRICT
RESUME
RETAIN
RETAINUPDATELOCKS
RETURN
RETURNED_SQLSTATE
RETURNING
RETURNS
REUSE
REVERSE
REVOKE
RIGHT
ROBIN
ROLE
ROLLBACK
ROLLFORWARD
ROLLING
ROOT
ROUND

ROUTINE
ROW
ROWID
ROWIDS
ROWNUMBER
ROWS
ROW_COUNT
ROW_NUMBER
RPAD
RTRIM
RULE

S

SAMEAS
SAMPLES
SAMPLING
SAVE
SAVEPOINT
SCHEMA
SCALE
SCROLL
SECLABEL_BY_COMP
SECLABEL_BY_NAME
SECLABEL_TO_CHAR
SECOND
SECONDARY
SECURED
SECURITY
SECTION
SELCONST
SELECT
SELECTING
SELECT_GRID
SELECT_GRID_ALL
SELFUNC
SELFUNCARGS
SENSITIVE
SEQUENCE
SERIAL
SERIAL8
SERIALIZABLE

SERVER
SERVER_NAME
SERVERUUID
SESSION
SET
SETSESSIONAUTH
SHARE
SHORT
SIBLINGS
SIGNED
SIN
SITENAME
SIZE
SKIP
SMALLFLOAT
SMALLINT
SOME
SOURCEID
SOURCETYPE
SPACE
SPECIFIC
SQL
SQLCODE
SQLCONTEXT
SQLERROR
SQLSTATE
SQLWARNING
SQRT
STABILITY
STACK
STANDARD
START
STAR_JOIN
STATCHANGE
STATEMENT
STATIC
STATISTICS
STATLEVEL
STATUS
STDEV
STEP
STOP

STORAGE
STORE
STRATEGIES
STRING
STRINGTOLABEL
STRUCT
STYLE
SUBCLASS_ORIGIN
SUBSTR
SUBSTRING
SUBSTRING_INDEX
SUM
SUPPORT
SYNC
SYNONYM
SYS*

T

T
TABLE
TABLES
TAN
TASK
TB
TEMP
TEMPLATE
TEST
TEXT
THEN
TIB
TIME
TO
TODAY
TO_CHAR
TO_DATE
TO_DSINTERVAL
TO_NUMBER
TO_YMINTERVAL
TRACE
TRAILING
TRANSACTION

TRANSITION
TREE
TRIGGER
TRIGGERS
TRIM
TRUE
TRUNC
TRUNCATE
TRUSTED
TYPE
TYPEDEF
TYPEID
TYPENAME
TYPEOF

U

UID
UNBOUNDED
UNCOMMITTED
UNDER
UNION
UNIQUE
UNIQUECHECK
UNITS
UNKNOWN
UNLOAD
UNLOCK
UNSIGNED
UPDATE
UPDATING
UPON
UPPER
USAGE
USE
USELASTCOMMITTED
USER
USE_DWA
USE_HASH
USE_NL
USING
USTLOW_SAMPLE

V

VALUE
VALUES
VAR
VARCHAR
VARIABLE
VARIANCE
VARIANT
VARYING
VERCOLS
VIEW
VIOLATIONS
VOID
VOLATILE

W - Z

WAIT
WARNING
WEEKDAY
WHEN
WHENEVER
WHERE
WHILE
WITH
WITHOUT
WORK
WRITE
WRITEDOWN
WRITEUP
XADATASOURCE
XID
XLOAD
XUNLOAD
YEAR

Guide to SQL: Reference

The *HCL® Informix® Guide to SQL: Reference* contains the reference information for the system catalog tables, data types, and environment variables of the HCL Informix® dialect of the SQL language, as implemented in HCL Informix®. These topics also include information about the stores_demo, sales_demo, and superstore_demo databases that are included with HCL Informix®.

This information is intended for the following users:

- Database users
- Database administrators
- Database security administrators
- Database application programmers.

This information assumes that you have the following background:

- A working knowledge of your computer, your operating system, and the utilities that your operating system provides.
- Some experience working with relational databases or exposure to database concepts.
- Some experience with computer programming.

System catalog tables

The *system catalog* consists of tables and views that describe the structure of the database. Sometimes called the *data dictionary*, these table objects contain everything that the database knows about itself. Each system catalog table contains information about specific elements in the database. Each database has its own system catalog.

These topics provide information about the structure, content, and use of the system catalog tables. It also contains information about the Information Schema, which provides information about the tables, views, and columns in all the databases of the HCL Informix® instance to which your user session is currently connected.

Objects That the System Catalog Tables Track

The system catalog tables maintain information about the database, including the following categories of database objects:

- Tables, views, synonyms, and table fragments
- Columns, constraints, indexes, and index fragments
- Distribution statistics for tables, indexes, and fragments
- Triggers on tables, and INSTEAD OF triggers on views
- Procedures, functions, routines, and associated messages
- Authorized users, roles, and privileges to access database objects
- LBAC security policies, components, labels, and exemptions
- Data types and casts
- User-defined aggregate functions
- Access methods and operator classes
- Sequence objects
- Storage spaces for BLOB and CLOB objects
- External optimizer directives
- Inheritance relationships
- XA data sources and XA data source types
- Trusted user and surrogate user information

Using the system catalog

HCL Informix® automatically generate the system catalog tables when you create a database. You can query the system catalog tables as you would query any other table in the database. The system catalog tables for a newly created database are located in a common area of the disk called a *dbspace*. Every database has its own system catalog tables. All tables and views in the system catalog have the prefix **sys** (for example, the **systables** system catalog table).

Not all tables with the prefix **sys** are true system catalog tables. For example, the **syscdr** database supports the Enterprise Replication feature. Non-catalog tables, however, have a **tabid** >= 100. System catalog tables all have a **tabid** < 100. See later in this section and SYSTABLES on page 1373 for more information about **tabid** numbers that the database server assigns to tables, views, synonyms, and (in HCL Informix®) sequence objects.

Tip: Do not confuse the system catalog tables of a database with the tables in the **sysmaster**, **sysutils**, **syscdr**, or (for HCL Informix®) the **sysadmin** and **sysuser** databases. The names of tables in those databases also have the **sys** prefix, but they contain information about an entire database server, which might manage multiple databases. Information in the **sysadmin**, **sysmaster**, **sysutils**, **syscdr**, and **sysuser** tables is primarily useful for database server administrators (DBSAs). See also the *HCL® Informix® Administrator's Guide* and *HCL® Informix® Administrator's Reference*.

The database server accesses the system catalog constantly. Each time an SQL statement is processed, the database server accesses the system catalog to determine system privileges, add or verify table or column names, and so on.

For example, the following CREATE SCHEMA block adds the **customer** table, with its indexes and privileges, to the **stores_demo** database. This block also adds a view, **california**, which restricts the data of the **customer** table to only the first and last names of the customer, the company name, and the telephone number for all customers who reside in California.

```
CREATE SCHEMA AUTHORIZATION maryl
CREATE TABLE customer (customer_num SERIAL(101), fname CHAR(15),
   lname CHAR(15), company CHAR(20), address1 CHAR(20), address2 CHAR(20),
   city CHAR(15), state CHAR(2), zipcode CHAR(5), phone CHAR(18))
GRANT ALTER, ALL ON customer TO cathl WITH GRANT OPTION AS maryl
GRANT SELECT ON customer TO public
GRANT UPDATE (fname, lname, phone) ON customer TO nhowe
CREATE VIEW california AS
   SELECT fname, lname, company, phone FROM customer WHERE state = 'CA'
CREATE UNIQUE INDEX c_num_ix ON customer (customer_num)
CREATE INDEX state_ix ON customer (state)
```

To process this CREATE SCHEMA block, the database server first accesses the system catalog to verify the following information:

- The new table and view names do not already exist in the database. (If the database is ANSI-compliant, the database server verifies that the new names do not already exist for the specified owners.)
- The user has permission to create tables and grant user privileges.
- The column names in the CREATE VIEW and CREATE INDEX statements exist in the **customer** table.

In addition to verifying this information and creating two new tables, the database server adds new rows to the following system catalog tables:

- **systables**
- **syscolumns**
- **sysviews**
- **systabauth**
- **syscolauth**
- **sysindexes**
- **sysindices**

Rows added to the systables system catalog table

The following two new rows of information are added to the **systables** system catalog table after the CREATE SCHEMA block is run.

Column name	First row	Second row
tabname	customer	california
owner	maryl	maryl
partnum	16778361	0
tabid	101	102
rowsize	134	134
ncols	10	4
nindexes	2	0
nrows	0	0
created	01/26/2007	01/26/2007
version	1	0
tabtype	T	V
locklevel	P	B
npused	0	0
fextsize	16	0
nextsize	16	0
flags	0	0
site		
dbname		

Each table recorded in the **systables** system catalog table is assigned a **tabid**, a system-assigned sequential number that uniquely identifies each table in the database. The system catalog tables receive 2-digit **tabid** numbers, and the user-created tables receive sequential **tabid** numbers that begin with `100`.

Rows added to the syscolumns system catalog table

The CREATE SCHEMA block adds 14 rows to the **syscolumns** system catalog table. These rows correspond to the columns in the table **customer** and the view **california**, as the following example shows.

colname	tabid	colno	coltype	collength	colmin	colmax
customer_num	101	1	262	4		
fname	101	2	0	15		
lname	101	3	0	15		
company	101	4	0	20		
address1	101	5	0	20		
address2	101	6	0	20		
city	101	7	0	15		
state	101	8	0	2		
zipcode	101	9	0	5		
phone	101	10	0	18		
fname	102	1	0	15		
lname	102	2	0	15		
company	102	3	0	20		
phone	102	4	0	18		

In the **syscolumns** table, each column within a table is assigned a sequential column number, **colno**, that uniquely identifies the column within its table. In the **colno** column, the **fname** column of the **customer** table is assigned the value `2` and the **fname** column of the view **california** is assigned the value `1`.

The **colmin** and **colmax** columns are empty. These columns contain values when a column is the first key (or the only key) in an index, has no NULL or duplicate values, and the UPDATE STATISTICS statement has been run.

Rows added to the sysviews system catalog table

The database server also adds rows to the **sysviews** system catalog table, whose **viewtext** column contains each line of the CREATE VIEW statement that defines the view. In that column, the **x0** that precedes the column names in the statement (for example, **x0.fname**) operates as an alias that distinguishes among the same columns that are used in a self-join.

Rows added to the systabauth system catalog table

The CREATE SCHEMA block also adds rows to the **systabauth** system catalog table. These rows correspond to the user privileges granted on **customer** and **california** tables, as the following example shows.

grantor	grantee	tabid	tabauth
maryl	public	101	su-idx--
maryl	cathl	101	SU-IDXAR
maryl	nhowe	101	--*-----
	maryl	102	SU-ID---

The **tabauth** column specifies the table-level privileges granted to users on the **customer** and **california** tables. This column uses an 8-byte pattern, such as `s` (Select), `u` (Update), `*` (column-level privilege), `i` (Insert), `d` (Delete), `x` (Index), `a` (Alter), and `r` (References), to identify the type of privilege. In this example, the user **nhowe** has column-level privileges on the **customer** table. A hyphen (-) means the user has not been granted the privilege whose position the hyphen occupies within the **tabauth** value.

If the **tabauth** privilege code is in uppercase (for example, `S` for Select), the user has this privilege and can also grant it to others; but if the privilege code is lowercase (for example, `s` for Select), the user cannot grant it to others.

Rows added to the syscolauth system catalog table

In addition, three rows are added to the **syscolauth** system catalog table. These rows correspond to the user privileges that are granted on specific columns in the **customer**, table as the following example shows.

grantor	grantee	tabid	colno	colauth
maryl	nhowe	101	2	-u-
maryl	nhowe	101	3	-u-
maryl	nhowe	101	10	-u-

The **colauth** column specifies the column-level privileges that are granted on the **customer** table. This column uses a 3-byte, pattern such as `s` (Select), `u` (Update), and `r` (References), to identify the type of privilege. For example, the user **nhowe** has Update privileges on the second column (because the **colno** value is `2`) of the **customer** table (indicated by **tabid** value of `101`).

Rows added to the sysindexes or the sysindices table

The CREATE SCHEMA block adds two rows to the **sysindexes** system catalog table (the **sysindices** table for HCL Informix®). These rows correspond to the indexes created on the **customer** table, as the following example shows.

idxname	c_num_ix	state_ix
owner	maryl	maryl
tabid	101	101
idxtype	U	D
clustered		
part1	1	8
part2	0	0
part3	0	0
part4	0	0
part5	0	0
part6	0	0
part7	0	0
part8	0	0
part9	0	0
part10	0	0
part11	0	0
part12	0	0
part13	0	0
part14	0	0
part15	0	0
part16	0	0
levels		
leaves		
nunique		
clust		
idxflags		

In this table, the **idxtype** column identifies whether the created index requires unique values (U) or accepts duplicate values (D). For example, the **c_num_ix** index on the **customer.customer_num** column is unique.

Accessing the system catalog

Normal user access to the system catalog is read-only. Users with Connect or Resource privileges cannot alter the catalog, but they can access data in the system catalog tables on a read-only basis using standard SELECT statements.

For example, the following SELECT statement displays all the table names and corresponding **tabid** codes of user-created tables in the database:

```
SELECT tabname, tabid FROM systables WHERE tabid > 99
```

When you use , only the tables that you created are displayed. To display the system catalog tables, enter the following statement:

```
SELECT tabname, tabid FROM systables WHERE tabid < 100
```

You can use the **SUBSTR** or the **SUBSTRING** function to select only part of a source string. To display the list of tables in columns, enter the following statement:

```
SELECT SUBSTR(tabname, 1, 18), tabid FROM systables
```

Although user **informix** can modify most system catalog tables, you should not update, delete, or insert any rows in them. Modifying the content of system catalog tables can affect the integrity of the database. However, you can safely use the ALTER TABLE statement to modify the size of the next extent of system catalog tables. Changing the next extent size does not affect extents that already exist.

For certain catalog tables of HCL Informix®, however, it is valid to add entries to the system catalog tables. For instance, in the case of the **syserrors** system catalog table and the **systracemsgs** system catalog table, a module developer can directly insert entries that are in these system catalog tables.

Update system catalog data

If you use the UPDATE STATISTICS statement to update the system catalog before executing a query or other data manipulation language (DML) statement, you can ensure that the information available to the query execution optimizer is current.

In HCL Informix®, the optimizer determines the most efficient strategy for executing SQL queries and other DML operations. The optimizer allows you to query the database without requiring you to consider fully which tables to search first in a join or which indexes to use. The optimizer uses information from the system catalog to determine the best query strategy.

When you delete or modify a table, the database server does not automatically update the related statistical data in the system catalog. For example, if you delete one or more rows in a table with the DELETE statement, the **nrows** column in the **systables** system catalog table, which holds the number of rows for that table, is not updated automatically.

The UPDATE STATISTICS statement causes the database server to recalculate data in the **systables**, **sysdistrib**, **syscolumns**, and **sysindices** system catalog tables, and in the **sysindexes** view. (For operations on fragmented tables where the STATLEVEL attribute is set to FRAGMENT, it also updates the **sysfragdist** and **sysfragments** system catalog tables.) After you run UPDATE STATISTICS, the **systables** system catalog table holds the correct value in the **nrows** column. If you specify

MEDIUM or HIGH mode when you run UPDATE STATISTICS, the **sysdistrib** and (for fragment-level statistics) the **sysfragdist** system catalog tables hold the updated column-distribution data.

Whenever you modify a data table extensively, use the UPDATE STATISTICS statement to update data in the system catalog. For more information about the UPDATE STATISTICS statement, see the *HCL® Informix® Guide to SQL: Syntax*.

Structure of the System Catalog

The following system catalog tables describe the database objects in a database.

System Catalog Tables
SYSAGGREGATES on page 1326
SYSAMS on page 1327
SYSATTRTYPES on page 1330
SYSAUTOLOCATE on page 1331
SYSBLOBS on page 1332
SYSCASTS on page 1332
SYSCHECKS on page 1333
SYSCHECKUDRDEP on page 1333
SYSCOLATTRIBS on page 1334
SYSCOLAUTH on page 1335
SYSCOLDEPEND on page 1335
SYSCOLUMNS on page 1336
SYSCONSTRAINTS on page 1341
SYSDEFAULTS on page 1342
SYSDEPEND on page 1343
SYSDIRECTIVES on page 1343
SYSDISTRIB on page 1344
SYSDOMAINS on page 1346
SYSERRORS on page 1346
#unique_1848
#unique_1849
#unique_1850

System Catalog Tables
SYSFRAGAUTH on page 1347
#unique_1852
SYSFRAGMENTS on page 1348
SYSINDEXES on page 1350
SYSINDICES on page 1352
SYSINHERITS on page 1355
SYSLANGAUTH on page 1356
SYSLOGMAP on page 1356
SYSOBJSTATE on page 1357
SYSOPCLASSES on page 1358
SYSPROCAUTH on page 1358
SYSPROCBODY on page 1359
SYSPROCCOLUMNS on page 1360
SYSPROCEDURES on page 1360
SYSPROCPLAN on page 1363
SYSREFERENCES on page 1364
SYSROLEAUTH on page 1365
SYSROUTINELANGS on page 1365
SYSSECLABELAUTH on page 1366
SYSSECLABELCOMPONENTS on page 1366
SYSSECLABELCOMPONENTELEMENTS on page 1366
SYSSECLABELNAMES on page 1367
SYSSECLABELS on page 1367
SYSSECPOLICIES on page 1367
SYSSECPOLICYCOMPONENTS on page 1368
SYSSECPOLICYEXEMPTIONS on page 1368
SYSSEQUENCES on page 1369
SYSSURROGATEAUTH on page 1370

System Catalog Tables
SYSSYNONYMS on page 1370
SYSSYNTABLE on page 1371
SYSTABAMDATA on page 1371
SYSTABAUTH on page 1372
SYSTABLES on page 1373
SYSTRACECLASSES on page 1376
SYSTRACEMSGS on page 1377
SYSTRIGBODY on page 1378
SYSTRIGGERS on page 1378
SYSUSERS on page 1379
SYSVIEWS on page 1380
SYSVIOLATIONS on page 1380
SYSXADATASOURCES on page 1381
SYSXASOURCETYPES on page 1381
SYSXTDDESC on page 1382
SYSXTDTYPEAUTH on page 1382
SYSXTDTYPES on page 1383

In case-sensitive databases that use the default database locale (U. S. English, ISO **8859-1** code set), character columns in these tables are CHAR and VARCHAR data types. For all other locales, character columns are the NLS data types, NCHAR and NVARCHAR. For information about differences in the collation order of character data types, see the *HCL® Informix® GLS User's Guide*. See also theData types on page 1389 chapter of this publication.

Character columns in databases that are not case-sensitive

In databases that are created with the NLSCASE INSENSITIVE keywords and that use the default database locale (U. S. English, ISO **8859-1** code set), character columns in system catalog tables are CHAR and VARCHAR data types, which support case-sensitive queries. For all other database locales, character column data types in the system catalog tables are the NLS data types, NCHAR and NVARCHAR, but with the following specific exceptions:

Table_name.Column_name	Data type
sysams.am_sptype	CHAR(3)
syscolauth.colauth	CHAR(3)

Table_name.Column_name	Data type
sysdefaults.class	CHAR(1)
sysfragauth.fragauth	CHAR(6)
sysinherits.class	CHAR(1)
syslangauth.langauth	CHAR(1)
sysprocauth.procauth	CHAR(1)
sysprocedures.mode	CHAR(1)
systabauth.tabauth	CHAR(9)
systriggers.event	CHAR(1)
sysxtdtypeauth.auth	CHAR(2)

In each of these columns, case-sensitive encoding can record information that utilities of the database server require in queries on those system catalog tables. In a database that is case-insensitive, queries might return incorrect results from data stored in NCHAR or NVARCHAR columns, if different attributes of database objects are encoded as different cases of the same letter. To avoid the loss of information, CHAR data types are used for the system catalog columns listed above.

SYSAGGREGATES

The **sysaggregates** system catalog table records user-defined aggregates (UDAs). The **sysaggregates** table has the following columns.

Table 31. SYSAGGREGATES table column descriptions

Column	Type	Explanation
name	VARCHAR(128)	Name of the aggregate
owner	CHAR(32)	Name of the owner of the aggregate
aggid	SERIAL	Unique code identifying the aggregate
init_func	VARCHAR(128)	Name of initialization UDR
iter_func	VARCHAR(128)	Name of iterator UDR
combine_func	VARCHAR(128)	Name of combine UDR

Table 31. SYSAGGREGATES table column descriptions

(continued)

Column	Type	Explanation
final_func	VARCHAR(128)	Name of finalization UDR
handlesnulls	BOOLEAN	NULL-handling indicator: • t = handles NULLs • f = does not handle NULLs

Each user-defined aggregate has one entry in **sysaggregates** that is uniquely identified by its identifying code (the **aggid** value). Only user-defined aggregates (aggregates that are not built in) have entries in **sysaggregates**.

Both a simple index on the **aggid** column and a composite index on the **name** and **owner** columns require unique values.

SYSAMS

The **sysams** system catalog table contains information that is required for using built-in access methods and those created by the CREATE ACCESS_METHOD statement of SQL.

The **sysams** table has the following columns.

Table 32. SYSAMS table column descriptions

Column	Type	Explanation
am_name	VARCHAR(128, 0)	Name of the access method
am_owner	CHAR(32)	Name of the owner of the access method
am_id	INTEGER	Unique identifying code for an access method This corresponds to the **am_id** columns in the **systables**, **sysindices**, and **sysopclasses** tables.
am_type	CHAR(1)	Type of access method: P = Primary; S = Secondary
am_sptype	CHAR(3)	Types of spaces where the access method can exist: • `A` means the access method supports extspaces and sbspaces. If the access method is built in, such as a B-tree, it also supports dbspaces. • `D` or `d` means the access method supports dbspaces only. • `DS` means the access method supports dbspaces and sbspaces.

Table 32. SYSAMS table column descriptions (continued)

Column	Type	Explanation
		• `S` or `s` means the access method supports sbspaces only. • `X` or `x` means the access method supports extspaces only. • `sx` means the access method supports sbspaces and extspaces.
am_defopclass	INTEGER	Unique identifying code for default-operator class Value is the **opclassid** from the entry for this operator class in the **sysopclasses** table.
am_keyscan	INTEGER	Whether a secondary access method supports a key scan (An access method supports a key scan if it can return a key and a rowid from a call to the **am_getnext** function.) (0 = FALSE; Non-zero = TRUE)
am_unique	INTEGER	Whether a secondary access method can support unique keys (0 = FALSE; Non-zero = TRUE)
am_cluster	INTEGER	Whether a primary access method supports clustering (0 = FALSE; Non-zero = TRUE)
am_rowids	INTEGER	Whether a primary access method supports rowids (0 = FALSE; Non-zero = TRUE)
am_readwrite	INTEGER	Whether a primary access method can both read and write (0 = access method is read-only; Non-zero = access method is read/write)
am_parallel	INTEGER	Whether an access method supports parallel execution (0 = FALSE; Non-zero = TRUE)
am_costfactor	SMALLFLOAT	The value to be multiplied by the cost of a scan to normalize it to costing done for built-in access methods The scan cost is the output of the **am_scancost** function.
am_create	INTEGER	The routine specified for the AM_CREATE purpose for this access method Value = **procid** for the routine in the **sysprocedures** table.
am_drop	INTEGER	The routine specified for the AM_DROP purpose function for this access method

Table 32. SYSAMS table column descriptions (continued)

Column	Type	Explanation
am_open	INTEGER	The routine specified for the AM_OPEN purpose function for this access method
am_close	INTEGER	The routine specified for the AM_CLOSE purpose function for this access method
am_insert	INTEGER	The routine specified for the AM_INSERT purpose function for this access method
am_delete	INTEGER	The routine specified for the AM_DELETE purpose function for this access method
am_update	INTEGER	The routine specified for the AM_UPDATE purpose function for this access method
am_stats	INTEGER	The routine specified for the AM_STATS purpose function for this access method
am_scancost	INTEGER	The routine specified for the AM_SCANCOST purpose function for this access method
am_check	INTEGER	The routine specified for the AM_CHECK purpose function for this access method
am_beginscan	INTEGER	The routine specified for the AM_BEGINSCAN purpose function for this access method
am_endscan	INTEGER	The routine specified for the AM_ENDSCAN purpose function for this access method
am_rescan	INTEGER	The routine specified for the AM_RESCAN purpose function for this access method
am_getnext	INTEGER	The routine specified for the AM_GETNEXT purpose function for this access method
am_getbyid	INTEGER	The routine specified for the AM_GETBYID purpose function for this access method
am_build	INTEGER	The routine specified for the AM_BUILD purpose function for this access method
am_init	INTEGER	The routine specified for the AM_INIT purpose function for this access method
am_truncate	INTEGER	The routine specified for the AM_TRUNCATE purpose function for this access method

Table 32. SYSAMS table column descriptions (continued)

Column	Type	Explanation
am_expr_pushdown	INTEGER	Whether parameter descriptors are supported (0 = FALSE; Non-zero = TRUE)

For each of the columns that contain a routine for a purpose function, the value is the **sysprocedures.procid** value for the corresponding routine.

A composite index on the **am_name** and **am_owner** columns in this table allows only unique values. The **am_id** column has a unique index.

For information about access method functions, see the documentation of your access method.

SYSATTRTYPES

The **sysattrtypes** system catalog table contains information about members of a complex data type. Each row of **sysattrtypes** contains information about elements of a collection data type or fields of a row data type.

The **sysattrtypes** table has the following columns.

Table 33. SYSATTRTYPES table column descriptions

Column	Type	Explanation
extended_id	INTEGER	Identifying code of an extended data type Value is the same as in the **sysxtdtypes** table (SYSXTDTYPES on page 1383).
seqno	SMALLINT	Identifying code of an entry having **extended_id** type
levelno	SMALLINT	Position of member in collection hierarchy
parent_no	SMALLINT	Value in the **seqno** column of the complex data type that contains this member
fieldname	VARCHAR(128)	Name of the field in a row type Null for other complex data types
fieldno	SMALLINT	Field number sequentially assigned by system (from left to right within each row type)
type	SMALLINT	Code for the data type See the description of **syscolumns.coltype** (page SYSCOLUMNS on page 1336).

Table 33. SYSATTRTYPES table column descriptions (continued)

Column	Type	Explanation
length	SMALLINT	Length (in bytes) of the member
xtd_type_id	INTEGER	Code identifying this data type See the description of **sysxtdtypes.extended_id** (SYSXTDTYPES on page 1383).

Two indexes on the **extended_id** column and the **xtd_type_id** column allow duplicate values. A composite index on the **extended_id** and **seqno** columns allows only unique values.

SYSAUTOLOCATE

The **sysautolocate** system catalog table indicates which dbspaces are available for automatic table fragmentation.

Table 34. SYSAUTOLOCATE table column descriptions

Column	Type	Explanation
dbsnum	INTEGER	The ID number of the dbspace. 0 indicates multiple dbspaces.
dbsname	VARCHAR(128,0)	The name of the dbspace. An asterisk (*) indicates multiple dbspaces.
pagesize	SMALLINT	The page size of the dbspace. 0 indicates multiple page sizes.
flags	INTEGER	• 1 = On. The dbspace is available for automatic table fragmentation. • 2 = Off. The dbspace is not available for automatic table fragmentation.

You add or remove dbspace from the list of available dbspace by running the task() or admin() SQL administration API function with one of the autolocate database arguments.

The **sysautolocate** system catalog table does not necessarily list every dbspace. For example, if all dbspaces are available for automatic table fragmentation, the table contains one row:

```
dbsnum      dbsname       pagesize     flags
0           *             0            1
```

If all but one dbspace is available, the table contains two rows, for example:

```
dbsnum      dbsname       pagesize     flags
0           *             0            1
12          dbs12         8            2
```

If all but two dbspaces are unavailable, the table contains three rows, for example:

```
dbsnum       dbsname        pagesize     flags
0            *              0            2
12           dbs12          8            1
13           dbs13          4            1
```

Related information

autolocate database argument: Specify dbspaces for automatic location and fragmentation (SQL administration API) on page

SYSBLOBS

The **sysblobs** system catalog table specifies the storage location of BYTE and TEXT column values. Its name is based on a legacy term for BYTE and TEXT columns, blobs (also known as *simple large objects*), and does not refer to the BLOB data type of HCL Informix®. The **sysblobs** table contains one row for each BYTE or TEXT column, and has the following columns.

Table 35. SYSBLOBS table column descriptions

Column	Type	Explanation
spacename	VARCHAR(128)	Name of partition, dbspace, or family
type	CHAR(1)	Code identifying the type of storage media: M = Magnetic
tabid	INTEGER	Code identifying the table
colno	SMALLINT	Column number within its table

A composite index on **tabid** and **colno** allows only unique values.

For information about the location and size of chunks of blobspaces, dbspaces, and sbspaces for TEXT, BYTE, BLOB, and CLOB columns, see the *HCL® Informix® Administrator's Guide* and the *HCL® Informix® Administrator's Reference*.

SYSCASTS

The **syscasts** system catalog table describes the casts in the database. It contains one row for each built-in cast, each implicit cast, and each explicit cast that a user defines. The **syscasts** table has the following columns.

Table 36. SYSCASTS table column descriptions

Column	Type	Explanation
owner	CHAR(32)	Owner of cast (user **informix** for built-in casts and *user* name for implicit and explicit casts)
argument_type	SMALLINT	Source data type on which the cast operates
argument_xid	INTEGER	Code for the source data type specified in the **argument_type** column
result_type	SMALLINT	Code for the data type returned by the cast

Table 36. SYSCASTS table column descriptions (continued)

Column	Type	Explanation
result_xid	INTEGER	Data type code of the data type named in the **result_type** column
routine_name	VARCHAR(128)	Function or procedure implementing the cast
routine_owner	CHAR(32)	Name of owner of the function or procedure specified in the **routine_name** column
class	CHAR(1)	Type of cast: E = Explicit cast I = Implicit cast S = Built-in cast

If **routine_name** and **routine_owner** have NULL values, this indicates that the cast is defined without a routine. This can occur if both of the data types specified in the **argument_type** and **result_type** columns have the same length and alignment, and are passed by reference, or passed by value.

A composite index on columns **argument_type**, **argument_xid**, **result_type**, and **result_xid** allows only unique values. A composite index on columns **result_type** and **result_xid** allows duplicate values.

SYSCHECKS

The **syschecks** system catalog table describes each check constraint defined in the database. Because the **syschecks** table stores both the ASCII text and a binary encoded form of the check constraint, it contains multiple rows for each check constraint. The **syschecks** table has the following columns.

Table 37. SYSCHECKS table column descriptions

Column	Type	Explanation
constrid	INTEGER	Unique code identifying the constraint
type	CHAR(1)	Form in which the check constraint is stored: B = Binary encoded s = Select T = Text
seqno	SMALLINT	Line number of the check constraint
checktext	CHAR(32)	Text of the check constraint

The text in the **checktext** column associated with `B` type in the type column is in computer-readable format. To view the text associated with a particular check constraint, use the following query with the appropriate **constrid** code:

```
SELECT * FROM syschecks WHERE constrid=10 AND type='T'
```

Each check constraint described in the **syschecks** table also has its own row in the **sysconstraints** table.

A composite index on the **constrid**, **type**, and **seqno** columns allows only unique values.

SYSCHECKUDRDEP

The **syscheckudrdep** system catalog table describes each check constraint that is referenced by a user-defined routine (UDR) in the database. The **syscheckudrdep** table has the following columns.

Table 38. SYSCHECKUDRDEP table column descriptions

Column	Type	Explanation
udr_id	INTEGER	Unique code identifying the UDR
constraint_id	INTEGER	Unique code identifying the check constraint

Each check constraint described in the **syscheckudrdep** table also has its own row in the **sysconstraints** system catalog table, where the **constrid** column has the same value as the **constraint_id** column of **syscheckudrdep**.

A composite index on the **udr_id** and **constraint_id** columns requires that combinations of these values be unique.

SYSCOLATTRIBS

The **syscolattribs** system catalog table describes the characteristics of smart large objects, namely CLOB and BLOB data types.

It contains one row for each sbspace referenced in the PUT clause of the CREATE TABLE statement or of the ALTER TABLE statement.

Table 39. SYSCOLATTRIBS table column descriptions

Column	Type	Explanation
tabid	INTEGER	Code uniquely identifying the table
colno	SMALLINT	Number of the column that contains the smart large object
extentsize	INTEGER	Pages in smart-large-object extent, expressed in KB
flags	INTEGER	Integer representation of the combination (by addition) of hexadecimal values of the following parameters: • LO_NOLOG (0x00000001 = 1) = The smart large object is not logged. • LO_LOG (0x00000010 = 2) = Logging of smart large objects conforms to current log mode of the database. • LO_KEEP_LASTACCESS_TIME (0x00000100 = 4) = Keeps a record of when this column was most recently accessed by a user. • LO_NOKEEP_LASTACCESS_TIME (0x00001000 = 8) = No record is kept of when this column was most recently accessed by a user.

Table 39. SYSCOLATTRIBS table column descriptions

(continued)

Column	Type	Explanation
		• HI_INTEG (0x00010000= 16) = Sbspace data pages have headers and footers to detect incomplete writes and data corruption. • MODERATE_INTEG (0x00100000= 32) = Data pages have headers but no footers.
flags1	INTEGER	Reserved for future use
sbspace	VARCHAR(128)	Name of the sbspace

A composite index on the **tabid**, **colno**, and **sbspace** columns allows only unique combinations of these values.

SYSCOLAUTH

The **syscolauth** system catalog table describes each set of discretionary access privileges granted on a column. It contains one row for each set of column-level privileges that are currently granted to a user, to a role, or to the PUBLIC group on a column in the database. The **syscolauth** table has the following columns.

Column	Type	Explanation
grantor	VARCHAR(32)	Authorization identifier of the grantor
grantee	VARCHAR(32)	Authorization identifier of the grantee
tabid	INTEGER	Code uniquely identifying the table
colno	SMALLINT	Column number within the table
colauth	CHAR(3)	3-byte pattern specifying column privileges: s *or* S = Select, u *or* U = Update, r *or* R = References

If the **colauth** privilege code is uppercase (for example, `S` for Select), a user who has this privilege can also grant it to others. If the **colauth** privilege code is lowercase (for example, `s` for Select), the user who has this privilege cannot grant it to others. A hyphen (`-`) indicates the absence of the privilege corresponding to that position within the **colauth** pattern.

A composite index on the **tabid**, **grantor**, **grantee**, and **colno** columns allows only unique values. A composite index on the **tabid** and **grantee** columns allows duplicate values.

SYSCOLDEPEND

The **syscoldepend** system catalog table tracks the table columns specified in check constraints and in NOT NULL constraints. Because a check constraint can involve more than one column in a table, the **syscoldepend** table can contain multiple rows for each check constraint; one row is created for each column involved in the constraint. The **syscoldepend** table has the following columns.

Column	Type	Explanation
constrid	INTEGER	Code uniquely identifying the constraint
tabid	INTEGER	Code uniquely identifying the table
colno	SMALLINT	Column number within the table

A composite index on the **constrid**, **tabid**, and **colno** columns allows only unique values. A composite index on the **tabid** and **colno** columns allows duplicate values.

See also the **syscheckudrdep** system catalog table in SYSCHECKUDRDEP on page 1333, which lists every check constraint that is referenced by a user-defined routine.

See also the **sysreferences** table in SYSREFERENCES on page 1364, which describes dependencies of referential constraints.

SYSCOLUMNS

The **syscolumns** system catalog table describes each column in the database.

One row exists for each column that is defined in a table or view.

Table 40. The SYSCOLUMNS table

Column	Type	Explanation
colname	VARCHAR(128)	Column name
tabid	INTEGER	Identifying code of table containing the column
colno	SMALLINT	Column number The system sequentially assigns this (from left to right within each table).
coltype	SMALLINT	Code indicating the data type of the column: 0 = CHAR 1 = SMALLINT 2 = INTEGER 3 = FLOAT 4 = SMALLFLOAT 5 = DECIMAL 6 = SERIAL [1] 7 = DATE 8 = MONEY 9 = NULL 10 = DATETIME

Table 40. The SYSCOLUMNS table (continued)

Column	Type	Explanation
		11 = BYTE 12 = TEXT 13 = VARCHAR 14 = INTERVAL 15 = NCHAR 16 = NVARCHAR 17 = INT8 18 = SERIAL8 [1] 19 = SET 20 = MULTISET 21 = LIST 22 = ROW (unnamed) 23 = COLLECTION 40 = LVARCHAR fixed-length opaque types [2] 41 = BLOB, BOOLEAN, CLOB variable-length opaque types [2] 43 = LVARCHAR (client-side only) 45 = BOOLEAN 52 = BIGINT 53 = BIGSERIAL [1] 2061 = IDSSECURITYLABEL [2, 3] 4118 = ROW (named)
collength	Any of the following data types: • Integer-based • Varying-length character • Time • Fixed-point • Simple-large-object • IDSSECURITYLABEL on page 1403	The value depends on the data type of the column. For some data types, the value is the column length (in bytes). See Storing Column Length on page 1339 for more information.
colmin	INTEGER	Holds the second-smallest value in a column
colmax	INTEGER	Holds the second-largest value in a column
extended_id	INTEGER	Data type code, from the **sysxtdtypes** table, of the data type specified in the **coltype** column

Table 40. The SYSCOLUMNS table (continued)

Column	Type	Explanation
seclabelid	INTEGER	The label ID of the security label associated with the column if it is a protected column. NULL otherwise.
colattr	SMALLINT	**HIDDEN** 1 - Hidden column **ROWVER** 2 - Row version column **ROW_CHKSUM** 4 - Row key column **ER_CHECKVER** 8 - ER row version column **UPGRD1_COL** 16 - ER auto primary key column **UPGRD2_COL** 32 - ER auto primary key column **UPGRD3_COL** 64 - ER auto primary key column **PK_NOTNULL** 128 - NOT NULL by PRIMARY KEY

Note:

[1] In , an offset value of 256 is always added to these **coltype** codes because sets SERIAL, SERIAL8, and BIGSERIAL columns to NOT NULL.
[2] The built-in opaque data types do not have a unique **coltype** value. They are distinguished by the **extended_id** column in the SYSXTDTYPES on page 1383 system catalog table.
[3] DISTINCT OF VARCHAR(128).

A composite index on **tabid** and **colno** allows only unique values.

The **coltype** codes can be incremented by bitmaps showing the following features of the column.

Bit Value	Significance When Bit Is Set
0x0100	NULL values are not allowed

Bit Value	Significance When Bit Is Set
0x0200	Value is from a host variable
0x0400	Float-to-decimal for networked database server
0x0800	DISTINCT data type
0x1000	Named ROW type
0x2000	DISTINCT type from LVARCHAR base type
0x4000	DISTINCT type from BOOLEAN base type
0x8000	Collection is processed on client system

For example, the **coltype** value `4118` for named row types is the decimal representation of the hexadecimal value `0x1016`, which is the same as the hexadecimal **coltype** value for an unnamed row type (`0x016`), with the named-row-type bit set. The file **$INFORMIXDIR/incl/esql/sqltypes.h** contains additional information about **syscolumns.coltype** codes.

The following table lists the **coltype** values for the built-in opaque data types:

NOT NULL constraints

Similarly, the **coltype** value is incremented by `256` if the column does not allow NULL values. To determine the data type for such columns, subtract `256` from the value and evaluate the remainder, based on the possible **coltype** values. For example, if the **coltype** value is `262`, subtracting `256` leaves a remainder of `6`, indicating that the column has a SERIAL data type.

Storing the column data type

The database server stores the **coltype** value as bitmap, as listed in SYSCOLUMNS on page 1336.

Storing column length

The **collength** column value depends on the data type of the column.

Integer-based data types

A **collength** value for a BIGINT, BIGSERIAL, DATE, INTEGER, INT8, SERIAL, SERIAL8, or SMALLINT column is machine-independent. The database server uses the following lengths for these integer-based data types of the SQL language.

Integer-based data types	Length (in bytes)
SMALLINT	2
DATE, INTEGER, and SERIAL	4
INT8 and SERIAL8	10
BIGINT and BIGSERIAL	8

Varying-length character data types

For HCL Informix® columns of the LVARCHAR type, **collength** has the value of *max* from the data type declaration, or `2048` if no maximum was specified.

For VARCHAR or NVARCHAR columns, the *max_size* and *min_space* values are encoded in the **collength** column using one of these formulas:

- If the **collength** value is positive:

  ```
  collength = (min_space * 256) + max_size
  ```

- If the **collength** value is negative:

  ```
  collength + 65536 = (min_space * 256) + max_size
  ```

Time data types

As noted previously, DATE columns have a value of `4` in the **collength** column.

For columns of type DATETIME or INTERVAL, **collength** is determined using the following formula:

```
(length * 256) + (first_qualifier * 16) + last_qualifier
```

The length is the physical length of the DATETIME or INTERVAL field, and *first_qualifier* and *last_qualifier* have values that the following table shows.

Field qualifier	Value	Field qualifier	Value
YEAR	0	FRACTION(1)	11
MONTH	2	FRACTION(2)	12
DAY	4	FRACTION(3)	13
HOUR	6	FRACTION(4)	14
MINUTE	8	FRACTION(5)	15
SECOND	10		

For example, if a DATETIME YEAR TO MINUTE column has a length of `12` (such as YYYY:DD:MO:HH:MI), a *first_qualifier* value of `0` (for YEAR), and a *last_qualifier* value of `8` (for MINUTE), then the **collength** value is `3080` (from `(256 * 12) + (0 * 16) + 8`).

Fixed-point data types

The **collength** value for a MONEY or DECIMAL (*p*, *s*) column can be calculated using the following formula:

```
(precision * 256) + scale
```

Simple-large-object data types

If the data type of the column is BYTE or TEXT, **collength** holds the length of the descriptor.

Storing Maximum and Minimum Values

The **colmin** and **colmax** are statistical values which give the second-smallest and second-largest values of a column at the time of the last update statistics. These values will be determined by update statistics for leading index columns only. For example, if the values in an indexed column are `1`, `2`, `3`, `4`, and `5`, the **colmin** value is `2` and the **colmax** value is `4`. Storing the second-smallest and second-largest data values lets the query optimizer make assumptions about the range of values in the column and, in turn, further refine search strategies.

The **colmin** and **colmax** columns contain values only if the column is indexed and the UPDATE STATISTICS statement has explicitly or implicitly calculated the column distribution. If you store BYTE or TEXT data in the tblspace, the **colmin** value is encoded as `-1`.

The **colmin** and **colmax** columns are valid only for data types that fit into four bytes: SMALLFLOAT, SMALLINT, INTEGER, and the first four bytes of CHAR. The values for all other noninteger column types are the initial four bytes of the maximum or minimum value, which are treated as integers.

It is better to use UPDATE STATISTICS MEDIUM than to depend on **colmin** and **colmax** values. UPDATE STATISTICS MEDIUM gives better information and is valid for all data types.

HCL Informix® does not calculate **colmin** and **colmax** values for user-defined data types. These columns, however, have values for user-defined data types if a user-defined secondary access method supplies them.

SYSCONSTRAINTS

The **sysconstraints** system catalog table lists the constraints placed on the columns in each database table. An entry is also placed in the **sysindexes** system catalog table (or **sysindices** view for HCL Informix®) for each unique, primary key, or referential constraint that does not already have a corresponding entry in **sysindexes** or **sysindices**. Because indexes can be shared, more than one constraint can be associated with an index. The **sysconstraints** table has the following columns.

Table 41. SYSCONSTRAINTS table column descriptions

Column	Type	Explanation
constrid	SERIAL	Code uniquely identifying the constraint
constrname	VARCHAR(128)	Name of the constraint
owner	VARCHAR(32)	Name of the owner of the constraint
tabid	INTEGER	Code uniquely identifying the table
constrtype	CHAR(1)	Code identifying the constraint type: • C = Check constraint • N = Not NULL • P = Primary key

Table 41. SYSCONSTRAINTS table column descriptions

(continued)

Column	Type	Explanation
		• R = Referential • T = Table • U = Unique
idxname	VARCHAR(128)	Name of index corresponding to constraint
collation	CHAR(32)	Collating order at the time when the constraint was created.

A composite index on the **constrname** and **owner** columns allows only unique values. An index on the **tabid** column allows duplicate values, and an index on the **constrid** column allows only unique values.

For check constraints (where **constrtype** = `c`), the **idxname** is always NULL. Additional information about each check constraint is contained in the **syschecks** and **syscoldepend** system catalog tables.

SYSDEFAULTS

The **sysdefaults** system catalog table lists the user-defined defaults that are placed on each column in the database. One row exists for each user-defined default value.

The **sysdefaults** table has the following columns:

Table 42. SYSDEFAULTS table column descriptions

Column	Type	Explanation
tabid	INTEGER	Code uniquely identifying a table. When the **class** column contains the code P, then the **tabid** column references a procedure ID not a table ID.
colno	SMALLINT	Code uniquely identifying a column.
type	CHAR(1)	Code identifying the type of default value: C = Current® L = Literal value N = NULL S = Dbservername *or* Sitename T = Today U = User
default	CHAR(256)	If **sysdefaults.type** = L, a literal default value.

Table 42. SYSDEFAULTS table column descriptions (continued)

Column	Type	Explanation
class	CHAR(1)	Code identifying what kind of column: `T` = table `t` = ROW type `P` = procedure

If no default is specified explicitly in the CREATE TABLE or the ALTER TABLE statement, then no entry exists for that column in the **sysdefaults** table.

If you specify a literal for the default value, it is stored in the **default** column as ASCII text. If the literal value is not of one of the data types listed in the next paragraph, the **default** column consists of two parts. The first part is the 6-bit representation of the binary value of the default value structure. The second part is the default value in ASCII text. A blank space separates the two parts.

If the data type of the column is not CHAR, NCHAR, NVARCHAR, or VARCHAR, or (for HCL Informix®) BOOLEAN or LVARCHAR, a binary representation of the default value is encoded in the **default** column.

A composite index on the **tabid**, **colno**, and **class** columns allows only unique values.

SYSDEPEND

The **sysdepend** system catalog table describes how each view or table depends on other views or tables. One row exists in this table for each dependency, so a view based on three tables has three rows. The **sysdepend** table has the following columns.

Table 43. SYSDEPEND table column descriptions

Column	Type	Explanation
btabid	INTEGER	Code uniquely identifying the base table or view
btype	CHAR(1)	Base object type: T = Table V = View
dtabid	INTEGER	Code uniquely identifying a dependent table or view
dtype	CHAR(1)	Code for the type of dependent object; currently, only view (V = View) is implemented

The **btabid** and **dtabid** columns are indexed and allow duplicate values.

SYSDIRECTIVES

The **sysdirectives** table stores external optimizer directives that can be applied to queries. Whether queries in client applications can use these optimizer directives depends on the setting of the **IFX_EXTDIRECTIVES** environment variable

on the client system, as described in Chapter 3, and on the EXT_DIRECTIVES setting in the configuration file of the database server.

The **sysdirectives** table has the following columns:

Table 44. SYSDIRECTIVES table column descriptions

Column	Type	Explanation
id	SERIAL	Unique code identifying the optimizer directive
query	TEXT	Text of the query as it exists in the application
directives	TEXT	Text of the optimizer directive, without comments
directive_code	BYTE	Encoded directive
active	SMALLINT	Integer code that identifies whether this entry is active (= 1) or test only (= 2)
hash_code	SMALLINT	For internal use only

NULL values are not valid in the **query** column. There is a unique index on the **id** column.

SYSDISTRIB

The **sysdistrib** system catalog table stores data-distribution information for the query optimizer to use. Data distributions provide detailed table and column information to the optimizer to improve the choice of execution paths of SELECT statements.

The **sysdistrib** table has the following columns.

Table 45. SYSDISTRIB table column descriptions

Column	Type	Explanation
tabid	INTEGER	Code identifying the table from which data values were gathered
colno	SMALLINT	Column number in the source table
seqno	INTEGER	Ordinal number for multiple entries
constructed	DATETIME YEAR TO FRACTION(5)	Date when the data distribution was created
mode	CHAR(1)	Optimization level: M = Medium H = High
resolution	SMALLFLOAT	Specified in the UPDATE STATISTICS statement
confidence	SMALLFLOAT	Specified in the UPDATE STATISTICS statement
encdat	STAT	Statistics information

Table 45. SYSDISTRIB table column descriptions (continued)

Column	Type	Explanation
type	CHAR(1)	Type of statistics: A = **encdat** has ASCII-encoded histogram in fixed-length character field S = **encdat** has user-defined statistics
smplsize	SMALLFLOAT	A value greater than zero up to 1.0 indicating a proportion of the total rows in the table that UPDATE STATISTICS samples. Values greater than 1.0 indicate the actual number of rows used that UPDATE STATISTICS samples. A value of zero indicates that no sample size is specified. UPDATE STATISTICS HIGH always updates statistics for all rows.
rowssmpld	FLOAT	Number of rows in the sample
constr_time	DATETIME YEAR TO FRACTION(5)	Time when the distribution was recorded
ustnrows	FLOAT	Rows in fragment when distribution was calculated.
ustbuildduration	INTERVAL HOUR TO FRACTION(5)	Time spent calculating the distribution statistics for this column
nupdates	FLOAT	Number of updates to the table
ndeletes	FLOAT	Number of deletes to the table
ninserts	FLOAT	Number of inserts to the table

Information is stored in the **sysdistrib** table when an UPDATE STATISTICS statement with mode MEDIUM or HIGH is executed for a table. (UPDATE STATISTICS LOW does not insert a value into the **mode** column.)

Only user **informix** can select the **encdat** column.

Each row in the **sysdistrib** system catalog table is keyed by the **tabid** and **colno** for which the statistics are collected.

For built-in data type columns, the **type** field is set to **A**. The **encdat** column stores an ASCII-encoded histogram that is broken down into multiple rows, each of which contains 256 bytes.

In HCL Informix®, for columns of user-defined data types, the **type** field is set to S. The **encdat** column stores the statistics collected by the **statcollect** user-defined routine in multirepresentational form. Only one row is stored for each **tabid** and **colno** pair. A composite index on the **tabid**, **colno**, and **seqno** columns requires unique combinations of values.

The following three DML counter columns record counts of how many DML operations modifying data rows were performed on the table at the time of generation of column distribution statistics:

- UPDATE operations in **nupdates**
- DELETE operations in **ndeletes**
- and INSERT operations in **ninserts**

These counts can also include rows modified by MERGE statements.

These DML counter columns store the values of the counters from the server partition that exists when distribution statistics are generated. If the AUTO_STAT_MODE configuration parameter, or the AUTO_STAT_MODE session environment setting, or the AUTO keyword of the UPDATE STATISTICS statement has enabled selective updating of data distribution statistics, the **ninserts**, **ndeletes**, and **ninserts** values can affect whether UPDATE STATISTICS operations refresh existing data distribution statistics. When the UPDATE STATISTICS statement runs in MEDIUM or HIGH mode against the table, the database server compares the stored values in these columns with the current values in the partition. Column distribution statistics for the table are not updated if the sum of the stored values differs from the sum of these current **sysdistrib** DML counter values from the partition page by less than the threshold specified by the setting of the STATCHANGE table attribute or of the STATCHANGE configuration parameter.

SYSDOMAINS

The **sysdomains** view is not used. It displays columns of other system catalog tables. It has the following columns.

Table 46. SYSDOMAINS table column descriptions

Column	Type	Explanation
id	SERIAL	Unique code identifying the domain
owner	CHAR(32)	Name of the owner of the domain
name	VARCHAR(128)	Name of the domain
type	SMALLINT	Code identifying the type of domain

There is no index on this view.

SYSERRORS

The **syserrors** system catalog table stores information about error, warning, and informational messages returned by modules and user-defined routines using the **mi_db_error_raise( )** function.

For a description of an error message, use the finderr utility.

The **syserrors** table has the following columns.

Column	Type	Explanation
sqlstate	CHAR(5)	SQLSTATE value associated with the error.

Column	Type	Explanation
locale	CHAR(36)	The locale with which this version of the message is associated (for example, **en_us.8859-1**)
level	SMALLINT	Reserved for future use
seqno	SMALLINT	Reserved for future use
message	VARCHAR(255)	Message text

To create a new message, insert a row directly into the **syserrors** table. By default, all users can view this table, but only users with the DBA privilege can modify it.

A composite index on the **sqlstate**, **locale**, **level**, and **seqno** columns allows only unique values.

Related information

Using the SQLSTATE Error Status Code on page

SYSFRAGAUTH

The **sysfragauth** system catalog table stores information about the privileges that are granted on table fragments. This table has the following columns.

Table 47. SYSFRAGAUTH table column descriptions

Column	Type	Explanation
grantor	CHAR(32)	Name of the grantor of privilege
grantee	CHAR(32)	Name of the grantee of privilege
tabid	INTEGER	Identifying code of the fragmented table
fragment	VARCHAR(128)	Name of dbspace where fragment is stored
fragauth	CHAR(6)	A 6-byte pattern specifying fragment privileges (including 3 bytes reserved for future use): • u *or* U = Update • i *or* I = Insert • d *or* D = Delete

In the **fragauth** column, an uppercase code (such as `U` for Update) means that the grantee can grant the privilege to other users; a lowercase (for example, `u` for Update) means the user cannot grant the privilege to others. Hyphen (-) indicates the absence of the privilege for that position within the pattern.

A composite index on the **tabid**, **grantor**, **grantee**, and **fragment** columns allows only unique values. A composite index on the **tabid** and **grantee** columns allows duplicate values.

The following example displays the fragment-level privileges for one base table, as they exist in the **sysfragauth** table. In this example, the grantee **rajesh** can grant the Update, Delete, and Insert privileges to other users.

grantor	grantee	tabid	fragment	fragauth
dba	omar	101	dbsp1	-ui---
dba	jane	101	dbsp3	--i---
dba	maria	101	dbsp4	--id--
dba	rajesh	101	dbsp2	-UID--

SYSFRAGMENTS

The **sysfragments** system catalog table stores fragmentation information and LOW mode statistical distributions for individual fragments of tables and indexes. One row exists for each table fragment or index fragment.

The **sysfragments** table has the following columns.

Column	Type	Explanation
fragtype	CHAR(1)	Code indicating the type of fragmented object: • I = Original index fragment • T = Original table fragment
tabid	INTEGER	Unique identifying code of table
indexname	VARCHAR(128)	Name of index
colno	INTEGER	Identifying code of TEXT or BYTE column, or the upper limit on the number of rolling window fragments
partn	INTEGER	Identifying code of physical storage location
strategy	CHAR(1)	Code for type of fragment distribution strategy: • R = Round-robin distribution strategy • E = Expression-based distribution strategy • I = IN DBSPACE clause specifies a storage location as part of distribution strategy

Column	Type	Explanation
		• N = raNge-iNterval (or rolliNg wiNdow) distribution strategy • L = List distribution strategy • T = Table-based distribution strategy • H = table is a subtable within a table Hierarchy
location	CHAR(1)	Reserved for future use; shows L for local
servername	VARCHAR(128)	Reserved for future use
evalpos	INTEGER	Position of fragment in the fragmentation list. For fragmentation by INTERVAL, one of the following values that indicates the type of information in the **exprtext** field: • -1 = List of dbspaces for interval fragments • -2 = Interval value • -3 = Fragmentation key • -4 = Rolling window fragment Fragmentation by LIST also uses the -3 value.
exprtext	TEXT	Expression for fragmentation strategy For fragmentation by INTERVAL, LIST, or rolling window, provides the information corresponding to the value of the **evalpos** field.
exprbin	BYTE	Binary version of expression
exprarr	BYTE	Range-partitioning data to optimize expression in range-expression fragmentation strategy
flags	INTEGER	Used internally
dbspace	VARCHAR(128)	Name of dbspace storing this fragment
levels	SMALLINT	Number of B-tree index levels
npused	FLOAT	For table-fragmentation strategies: the number of data pages For index-fragmentation strategies: the number of leaf pages For rolling window tables: the units for the storage size limit in nrows
nrows	FLOAT	For tables: the number of rows in the fragment. For indexes: the number of unique keys.

Column	Type	Explanation
		For rolling window tables: the upper limit on storage size in the purge policy.
clust	FLOAT	Degree of index clustering; smaller numbers correspond to greater clustering.
partition	VARCHAR(128)	Fragment name.This can match the name of the dbspace that stores the fragment, or can be an arbitrary name.
version	SMALLINT	Number that increments when fragment statistics is updated
nupdates	FLOAT	Number of updates to the fragment
ndeletes	FLOAT	Number of deletes to the fragment
ninserts	FLOAT	Number of inserts to the fragment

Every fragment has a row in this table. The **evalpos** and **evaltext** fields contain information about individual fragments.

Tables and indexes created with fragmentation by INTERVAL or LIST have additional rows containing information about the fragmentation strategy.

The **strategy** type `T` is used for attached indexes. (This is a fragmented index whose fragmentation strategy is the same as for the table fragmentation.)

For information about the **nupdates**, **ndeletes**, and **ninserts** columns, which in **sysfragments** tabulate DML operations on a table since the most recent recalculation of its distribution statistics, see the description of the three columns that have the same names in the SYSDISTRIB on page 1344 system catalog table.

In Informix®, a composite index on the **fragtype**, **tabid**, **indexname**, and **evalpos** columns allows duplicate values.

SYSINDEXES

The **sysindexes** table is a view on the **sysindices** table. It contains one row for each index in the database.

The **sysindexes** table has the following columns.

Table 48. SYSINDEXES table column descriptions

Column	Type	Explanation
idxname	VARCHAR(128)	Index name
owner	VARCHAR(32)	Owner of index (user **informix** for system catalog tables and *username* for database tables)
tabid	INTEGER	Unique identifying code of table

Table 48. SYSINDEXES table column descriptions

(continued)

Column	Type	Explanation
idxtype	CHAR(1)	Index type: U = Unique D = Duplicates allowed G = Nonbitmap generalized-key index g = Bitmap generalized-key index u = unique, bitmap d = nonunique, bitmap
clustered	CHAR(1)	Clustered or nonclustered index (C = Clustered)
part1	SMALLINT	Column number (**colno**) of a single index or the 1st component of a composite index
part2	SMALLINT	2nd component of a composite index
part3	SMALLINT	3rd component of a composite index
part4	SMALLINT	4th component of a composite index
part5	SMALLINT	5th component of a composite index
part6	SMALLINT	6th component of a composite index
part7	SMALLINT	7th component of a composite index
part8	SMALLINT	8th component of a composite index
part9	SMALLINT	9th component of a composite index
part10	SMALLINT	10th component of a composite index
part11	SMALLINT	11th component of a composite index
part12	SMALLINT	12th component of a composite index
part13	SMALLINT	13th component of a composite index
part14	SMALLINT	14th component of a composite index
part15	SMALLINT	15th component of a composite index
part16	SMALLINT	16th component of a composite index
levels	SMALLINT	Number of B-tree levels
leaves	INTEGER	Number of leaves
nunique	INTEGER	Number of unique keys in the first column

Table 48. SYSINDEXES table column descriptions

(continued)

Column	Type	Explanation
clust	INTEGER	Degree of clustering; smaller numbers correspond to greater clustering
idxflags	INTEGER	Bitmap storing the current locking mode of the index

As with most system catalog tables, changes that affect existing indexes are reflected in this table only after you run the UPDATE STATISTICS statement.

Each **part1** through **part16** column in this table holds the column number (**colno**) of one of the 16 possible parts of a composite index. If the component is ordered in descending order, the **colno** is entered as a negative value. The columns are filled in for B-tree indexes that do not use user-defined data types or functional indexes. For generic B-trees and all other access methods, the **part1** through **part16** columns all contain zeros.

The **clust** column is blank until the UPDATE STATISTICS statement is run on the table. The maximum value is the number of rows in the table, and the minimum value is the number of data pages in the table.

SYSINDICES

The **sysindices** system catalog table describes the indexes in the database. It stores LOW mode statistics for all indexes, and contains one row for each index that is defined in the database.

Table 49. sysindices system catalog table columns

Column	Type	Explanation
idxname	VARCHAR(128)	Name of index
owner	VARCHAR(32)	Name of owner of index (user **informix** for system catalog tables and *username* for database tables)
tabid	INTEGER	Unique identifying code of table
idxtype	CHAR(1)	Uniqueness status U = Unique values required D = Duplicates allowed
clustered	CHAR(1)	Clustered or nonclustered status (C = Clustered)

Table 49. sysindices system catalog table columns

(continued)

Column	Type	Explanation
levels	SMALLINT	Number of tree levels
leaves	FLOAT	Number of leaves
nunique	FLOAT	Number of unique keys in the first column
clust	FLOAT	Degree of clustering; smaller numbers correspond to greater clustering. The maximum value is the number of rows in the table, and the minimum value is the number of data pages in the table. This column is blank until UPDATE STATISTICS is run on the table.
nrows	FLOAT	Estimated number of rows in the table (zero until UPDATE STATISTICS is run on the table)
indexkeys	INDEXKEYARRAY	Internal representation of the index keys. Column can have up to three fields, in the format: **procid, (***col1*,*col2*, . . . , *coln***), opclassid** where 1 < *n* < 341
amid	INTEGER	Unique identifying code of the access method that implements this index. (Value = **am_id** for that access method in the **sysams** table.)
ampparam	LVARCHAR(2048)	List of parameters used to customize the **amid** access method behavior
collation	CHAR(32)	Database locale whose collating order was in effect at the time of index creation
pagesize	INTEGER	Size of the page, in bytes, where this index is stored
nhashcols	SMALLINT	Number of hashed columns in a FOT index
nbuckets	SMALLINT	Number of subtrees (buckets) in a forest of trees (FOT) index
ustlowts	DATETIME YEAR TO FRACTION	Date and time when index statistics were last recorded
ustbuilddu	INTERVAL HOUR TO	Time required to calculate index statistics

Table 49. sysindices system catalog table columns

(continued)

Column	Type	Explanation
ration	FRACTION (5)	
nupdates	FLOAT	Number of updates to the table
ndeletes	FLOAT	Number of deletes to the table
ninserts	FLOAT	Number of inserts to the table
fextsize	INT	Size (in KB) of the first extent of the index
nextsize	INT	Size (in KB) of the next extent of the index
indexattr	INT	• 0x00000001 = The index has a partial column key • 0x00000002 = The index is compressed • 0x00000004 = The index is on a BSON column • 0x000000010 = The index is invisible
jparam	LVARCHAR(2048)	BSON index information

Tip: This system catalog table is changed from Version 7.2 of HCL Informix®. The earlier schema of this system catalog table is still available as a view that can be accessed under its original name: **sysindexes**. See SYSINDEXES on page 1350.

Changes that affect existing indexes are reflected in this system catalog table only after you run the UPDATE STATISTICS statement.

The fields within the **indexkeys** columns have the following significance:

- The **procid** (as in **sysprocedures**) exists only for a functional index on return values of a function defined on columns of the table.
- The list of columns (*col1*, *col2*, ... , *coln*) in the second field identifies the columns on which the index is defined. The maximum is language-dependent: up to 341 for an SPL or Java™ UDR; up to 102 for a C UDR.
- The **opclassid** identifies the secondary access method that the database server used to build and to search the index. This is the same as the **sysopclasses.opclassid** value for the access method.

For information about the **nupdates**, **ndeletes**, and **ninserts** columns, which in **sysindices** tabulate DML operations on an index since the most recent recalculation of its distribution statistics, see the description of the three columns that have the same names in the SYSDISTRIB on page 1344 system catalog table.

The **fextsize** column shows the user-defined first extent size (in kilobytes) that the optional EXTENT SIZE clause specified in the CREATE INDEX statement that defined the index. Similarly, the **nextsize** column shows the user-defined next extent size (in kilobytes) that the optional NEXT SIZE clause specified in the CREATE INDEX statement. Each of these columns displays a value of zero (`0`) if the corresponding EXTENT SIZE or NEXT SIZE clause was omitted when the index was created.

If the CREATE INDEX statement that defines a new index includes no explicit extent size specifications, the database server automatically calculates the first and next extent sizes, but the **fextsize** and **nextsize** column values are set to `0`. When the database server is converted from a release earlier than Version 11.70, the **fextsize** and **nextsize** values for every migrated index are `0`.

The **tabid** column is indexed and allows duplicate values. A composite index on the **idxname**, **owner**, and **tabid** columns allows only unique values.

Related information

VISIBLE and INVISIBLE options for indexes

SYSINHERITS

The **sysinherits** system catalog table stores information about table hierarchies and named ROW type inheritance. Every supertype, subtype, supertable, and subtable in the database has a corresponding row in the **sysinherits** table.

Column	Type	Explanation
child	INTEGER	Identifying code of the subtable or subtype
parent	INTEGER	Identifying code of the supertable or supertype
class	CHAR(1)	Inheritance class: `t` = named ROW type `T` = table

The **child** and **parent** values are from **sysxtdtypes.extended_id** for named ROW types, or from **systables.tabid** for tables. Simple indexes on the **child** and **parent** columns allow duplicate values.

SYSLANGAUTH

The **syslangauth** system catalog table contains the authorization information about computer languages that are used to write user-defined routines (UDRs).

Table 50. SYSLANGAUTH table column descriptions

Column	Type	Explanation
grantor	VARCHAR(32)	Name of the grantor of the language authorization
grantee	VARCHAR(32)	Name of the grantee of the language authorization
langid	INTEGER	Identifying code of language in **sysroutinelangs** table
langauth	CHAR(1)	The language authorization: u = Usage privilege granted U = Usage privilege granted WITH GRANT OPTION

A composite index on the **langid**, **grantor**, and **grantee** columns allows only unique values. A composite index on the **langid** and **grantee** columns allows duplicate values.

SYSLOGMAP

The **syslogmap** system catalog table contains fragmentation information.

Table 51. SYSLOGMAP table column descriptions

Column	Type	Explanation
tabloc	INTEGER	Code for the location of a table in another database
tabid	INTEGER	Unique identifying code of the table
fragid	INTEGER	Identifying code of the fragment
flags	INTEGER	Bitmap of modifiers from declaration of fragment

A simple index on the **tabloc** column and a composite index on the **tabid** and **fragid** columns do not allow duplicate values.

SYSOBJSTATE

The **sysobjstate** system catalog table stores information about the state (object mode) of database objects. The types of database objects that are listed in this table are indexes, triggers, and constraints.

Every index, trigger, and constraint in the database has a corresponding row in the **sysobjstate** table if a user creates the object. Indexes that the database server creates on the system catalog tables are not listed in the **sysobjstate** table because their object mode cannot be changed.

The **sysobjstate** table has the following columns.

Table 52. SYSOBJSTATE table column descriptions

Column	Type	Explanation
objtype	CHAR(1)	Code for the type of database object: • C = Constraint • I = Index • T = Trigger
owner	VARCHAR(32)	Authorization identifier of the owner of the database object
name	VARCHAR(128)	Name of the database object
tabid	INTEGER	Identifying code of table on which the object is defined
state	CHAR(1)	The current state (object mode) of the database object. This value can be one of the following codes: • D = Disabled • E = Enabled • F = Filtering with no integrity-violation errors • G = Filtering with integrity-violation error
visible	CHAR(1)	The current state(visibility) of an index. Visibility is applicable only to indexes. Possible values include: • 'Y' (Yes) = Visible • 'N' (No) = Invisible • '-' (NA) = Not applicable

A composite index on the **objtype**, **name**, **owner**, and **tabid** columns allows only unique combinations of values. A simple index on the **tabid** column allows duplicate values.

Related information

VISIBLE and INVISIBLE options for indexes

SYSOPCLASSES

The **sysopclasses** system catalog table contains information about operator classes associated with secondary access methods. It contains one row for each operator class that has been defined in the database. The **sysopclasses** table has the following columns.

Column	Type	Explanation
opclassname	VARCHAR(128)	Name of the operator class
owner	VARCHAR(32)	Name of the owner of the operator class
amid	INTEGER	Identifying code of the secondary access method associated with this operator class
opclassid	SERIAL	Identifying code of the operator class
ops	LVARCHAR(2048)	List of names of the operators that belong to this operator class
support	LVARCHAR(2048)	List of names of support functions defined for this operator class

The **opclassid** value corresponds to the **sysams.am_defopclass** value that specifies the default operator class for the secondary access method that the **amid** column specifies.

The **sysopclasses** table has a composite index on the **opclassname** and **owner** columns and an index on **opclassid** column. Both indexes allow only unique values.

SYSPROCAUTH

The **sysprocauth** system catalog table describes the privileges granted on a procedure or function. It contains one row for each set of privileges that is granted. The **sysprocauth** table has the following columns.

Table 53. SYSPROCAUTH table column descriptions

Column	Type	Explanation
grantor	VARCHAR(32)	Name of grantor of privileges to access the routine

Table 53. SYSPROCAUTH table column descriptions

(continued)

Column	Type	Explanation
grantee	VARCHAR(32)	Name of grantee of privileges to access the routine
procid	INTEGER	Unique identifying code of the routine
procauth	CHAR(1)	Type of privilege granted on the routine: e = Execute privilege on routine E = Execute privilege WITH GRANT OPTION

A composite index on the **procid**, **grantor**, and **grantee** columns allows only unique values. A composite index on the **procid** and **grantee** columns allows duplicate values.

SYSPROCBODY

The **sysprocbody** system catalog table describes the compiled version of each procedure or function in the database. Because the **sysprocbody** table stores the text of the routine, each routine can have multiple rows. The **sysprocbody** table has the following columns.

Table 54. SYSPROCBODY table column descriptions

Column	Type	Explanation
procid	INTEGER	Unique identifying code for the routine
datakey	CHAR(1)	Type of information in the **data** column: A = Routine alter SQL (will not change this value after update statistics) D = Routine user documentation text E = Time of creation information L = Literal value (that is, literal number or quoted string) P = Interpreter instruction code (p-code) R = Routine return value type list S = Routine symbol table T = Routine text creation SQL
seqno	INTEGER	Line number within the routine
data	CHAR(256)	Actual text of the routine

The A flag indicates the procedure modifiers are altered. ALTER ROUTINE statement updates only modifiers and not the routine body. UPDATE STATISTICS updates the query plan and not the routine modifiers, and the value of datakey will not be changed from A. The A flag marks all the procedures and functions that have altered modifiers, including overloaded procedures and functions. The T flag is used for routine creation text.

The **data** column contains actual data, which can be in one of these formats:

- Encoded return values list
- Encoded symbol table
- Literal data
- P-code for the routine
- Compiled code for the routine
- Text of the routine and its documentation

A composite index on the **procid**, **datakey**, and **seqno** columns allows only unique values.

SYSPROCCOLUMNS

The **sysproccolumns** system catalog table stores information about return types and parameter names of all UDRs in SYSPROCEDURES.

A composite index on the **procid** and **paramid** columns in this table allows only unique values.

Table 55. SYSPROCCOLUMNS table column descriptions

Column	Type	Explanation
procid	INTEGER	Unique identifying code of the routine
paramid	INTEGER	Unique identifying code of the parameter
paramname	VARCHAR (IDENTSIZE)	Name of the parameter
paramtype	SMALLINT	Identifies the type of parameter
paramlen	SMALLINT	Specifies the length of the parameter
paramxid	INTEGER	Specifies the extended type ID for the parameter
paramattr	INTEGER	0 = Parameter is of unknown type 1 = Parameter is INPUT mode 2 = Parameter is INOUT mode 3 = Parameter is multiple return value 4 = Parameter is OUT mode 5 = Parameter is a return value

SYSPROCEDURES

The **sysprocedures** system catalog table lists the characteristics for each function and procedure that is registered in the database. It contains one row for each routine.

Each function in **sysprocedures** has a unique value, **procid**, called a *routine identifier*. Throughout the system catalog, a function is identified by its routine identifier, not by its name.

The **sysprocedures** table has the following columns.

Table 56. SYSPROCEDURES table column descriptions

Column	Type	Explanation
procname	VARCHAR(128)	Name of routine
owner	VARCHAR(32)	Name of owner
procid	SERIAL	Unique identifying code for the routine
mode	CHAR(1)	Mode type: D *or* d = DBA O *or* o = Owner P *or* p = Protected R *or* r = Restricted T *or* t = Trigger
retsize	INTEGER	Compiled size (in bytes) of returned values
symsize	INTEGER	Compiled size (in bytes) of symbol table
datasize	INTEGER	Compiled size (in bytes) of constant data
codesize	INTEGER	Compiled size (in bytes) of routine code
numargs	INTEGER	Number of arguments to routine
isproc	CHAR(1)	Specifies if the routine is a procedure or a function: `t` = procedure `f` = function
specificname	VARCHAR(128)	Specific name for the routine
externalname	VARCHAR(255)	Location of the external routine. This item is language-specific in content and format.
paramstyle	CHAR(1)	Parameter style: `I` = HCL Informix®
langid	INTEGER	Language code (in **sysroutinelangs** table)
paramtypes	RTNPARAMTYPES	Information describing the parameters of the routine
variant	BOOLEAN	Whether the routine is VARIANT or not:

Table 56. SYSPROCEDURES table column descriptions (continued)

Column	Type	Explanation
		t = is VARIANT f = is not VARIANT
client	BOOLEAN	Reserved for future use
handlesnulls	BOOLEAN	NULL handling indicator: t = handles NULLs f = does not handle NULLs
percallcost	INTEGER	Amount of CPU per call Integer cost to execute UDR: cost/call - 0 -(2^31-1)
commutator	VARCHAR(128)	Name of commutator function
negator	VARCHAR(128)	Name of the negator function
selfunc	VARCHAR(128)	Name of function to estimate selectivity of the UDR
internal	BOOLEAN	Specifies if the routine can be called from SQL: t = routine is internal, not callable from SQL f = routine is external, callable from SQL
class	CHAR(18)	CPU class by which the routine should be executed
stack	INTEGER	Stack size in bytes required per invocation
parallelizable	BOOLEAN	Parallelization indicator for UDR: t = parallelizable f = not parallelizable
costfunc	VARCHAR(128)	Name of the cost function for the UDR
selconst	SMALLFLOAT	Selectivity constant for UDR
procflags	INTEGER	For internal use only
collation	CHAR(32)	Collating order at the time when the routine was created

In the **mode** column, the R mode is a special case of the O mode. A routine is in restricted (R) mode if it was created with a specified owner who is different from the routine creator. If routine statements involving a remote database are executed,

the database server uses the access privileges of the user who executes the routine instead of the privileges of the routine owner. In all other scenarios, R-mode routines behave the same as O-mode routines.

The database server can create protected routines for internal use. The **sysprocedures** table identifies these protected routines with the letter `P` or `p` in the **mode** column, where `p` indicates an SPL routine. Protected routines have the following restrictions:

- You cannot use the ALTER FUNCTION, ALTER PROCEDURE, or ALTER ROUTINE statements to modify protected routines.
- You cannot use the DROP FUNCTION, DROP PROCEDURE, or DROP ROUTINE statements to unregister protected routines.
- You cannot use the dbschema utility to display protected routines.

In earlier versions, protected SPL routines were indicated by a lowercase `p`. Starting with version 9.0, protected SPL routines are treated as DBA routines and cannot be Owner routines. Thus `D` and `O` indicate DBA routines and Owner routines, while `d` and `o` indicate protected DBA routines and protected Owner routines.

The trigger mode designates user-defined SPL routines that can be invoked only from the FOR EACH ROW section of a triggered action.

Important: After you issue the SET SESSION AUTHORIZATION statement, the database server assigns a restricted mode to all Owner routines that you created while using the new identity.

A unique index is defined on the **procid** column. A composite index on the **procname**, **isproc**, **numargs**, and **owner** columns allows duplicate values, as does a composite index on the **specificname** and **owner** columns.

SYSPROCPLAN

The **sysprocplan** system catalog table describes the query-execution plans and dependency lists for data-manipulation statements within each routine. Because different parts of a routine plan can be created on different dates, this table can contain multiple rows for each routine.

Table 57. SYSPROCPLAN table column descriptions

Column	Type	Explanation
procid	INTEGER	Identifying code for the routine
planid	INTEGER	Identifying code for the plan
datakey	CHAR(1)	Type of information stored in **data** column: D = Dependency list I = Information record Q = Execution plan

Table 57. SYSPROCPLAN table column descriptions (continued)

Column	Type	Explanation
seqno	INTEGER	Line number within the plan
created	DATE	Date when plan was created
datasize	INTEGER	Size (in bytes) of the list or plan
data	CHAR(256)	Encoded (compiled) list or plan

Before a routine is run, its dependency list in the **data** column is examined. If the major version number of a table accessed by the plan has changed, or if any object that the routine uses has been modified since the plan was optimized (for example, if an index has been dropped), then the plan is optimized again. When **datakey** is I, the **data** column stores information about UPDATE STATISTICS and PDQPRIORITY.

It is possible to delete all the plans for a given routine by using the DELETE statement on **sysprocplan**. When the routine is subsequently executed, new plans are automatically generated and recorded in **sysprocplan**. The UPDATE STATISTICS FOR PROCEDURE statement also updates this table.

A composite index on the **procid**, **planid**, **datakey**, and **seqno** columns allows only unique values.

SYSREFERENCES

The **sysreferences** system catalog table lists all referential constraints on columns. It contains a row for each referential constraint in the database.

Table 58. SYSREFERENCES table column descriptions

Column	Type	Explanation
constrid	INTEGER	Code uniquely identifying the constraint
primary	INTEGER	Identifying code of the corresponding primary key
ptabid	INTEGER	Identifying code of the table that is the primary key
updrule	CHAR(1)	Reserved for future use; displays an R
delrule	CHAR(1)	Whether constraint uses cascading delete or restrict rule: C = Cascading delete R = Restrict (default)
matchtype	CHAR(1)	Reserved for future use; displays an N
pendant	CHAR(1)	Reserved for future use; displays an N

The **constrid** column is indexed and allows only unique values. The **primary** column is indexed and allows duplicate values.

SYSROLEAUTH

The **sysroleauth** system catalog table describes the roles that are granted to users. It contains one row for each role that is granted to a user in the database. The **sysroleauth** table has the following columns.

Table 59. SYSROLEAUTH table column descriptions

Column	Type	Explanation
rolename	VARCHAR(32)	Name of the role
grantee	VARCHAR(32)	Name of the grantee of the role
is_grantable	CHAR(1)	Specifies whether the role is grantable: Y = Grantable N = Not grantable

The **is_grantable** column indicates whether the role was granted with the WITH GRANT OPTION of the GRANT statement.

A composite index on the **rolename** and **grantee** columns allows only unique values.

SYSROUTINELANGS

The **sysroutinelangs** system catalog table lists the supported programming languages for user-defined routines (UDRs). It has these columns.

Column	Type	Explanation
langid	SERIAL	Code uniquely identifying a supported language
langname	CHAR(30)	Name of the language, such as C or SPL
langinitfunc	VARCHAR(128)	Name of initialization function for the language
langpath	CHAR(255)	Directory path for the UDR language
langclass	CHAR(18)	Name of the class of the UDR language

An index on the **langname** column allows duplicate values.

SYSSECLABELAUTH

The **sysseclabelauth** system catalog table records the LBAC labels that have been granted to users. It has these columns.

Column	Type	Explanation
GRANTEE	CHAR(32)	The name of the label grantee
secpolicyid	INTEGER	The ID of the security policy to which the security label belongs.
readseclabelid	INTEGER	The security label ID of the security label granted for read access
writeseclabelid	INTEGER	The security label ID of the security label granted for write access

SYSSECLABELCOMPONENTS

The **sysseclabelcomponents** system catalog table records security label components. It has these columns.

Column	Type	Explanation
compname	VARCHAR(128)	Component name
compid	SERIAL	Component ID
comptype	CHAR(1)	The component type: A = array S = set T = tree
numelements	INTEGER	Number of elements in the component
coveringinfo	VARCHAR(128)	Internal encoding information
numalters	SMALLINT	Numbers of alter operations that have been performed on the component

SYSSECLABELCOMPONENTELEMENTS

The **sysseclabelcomponentelements** system catalog table records the values of component elements of security labels. It has these columns.

Column	Type	Explanation
compid	INTEGER	Component ID
element	VARCHAR(32)	Element name

Column	Type	Explanation
elementencoding	CHAR(8)	Encoded form of the element
parentelement	VARCHAR(32)	The name of the parent elements for tree components. The value is NULL for the following items: Set components Array components Root nodes of a tree component
alterversion	SMALLINT	The number of the alter operation when the element is added. This value is used by the **dbexport** and **dbimport** commands.

SYSSECLABELNAMES

The **sysseclabelnames** system catalog table records the security label names. It has these columns.

Column	Type	Explanation
secpolicyid	INTEGER	The ID of the security policy to which the security label belongs.
seclabelname	VARCHAR(128)	The name of the security label
seclabelid	INTEGER	The ID of the security label

SYSSECLABELS

The **sysseclabels** system catalog table records the security label encoding. It has these columns.

Column	Type	Explanation
secpolicyid	INTEGER	ID of the security policy to which the security label belongs
seclabelid	INTEGER	Security label ID
sysseclabelnames	VARCHAR(128)	Security label encoding

SYSSECPOLICIES

The **syssecpolicies** system catalog table records security policies It has these columns.

Column	Type	Explanation
secpolicyname	VARCHAR(128)	Security policy name
secpolicyid	SERIAL	Security policy ID

Column	Type	Explanation
numcomps	SMALLINT	Number of security label components in the security policy
comptypelist	CHAR(16)	An ordered list of the type of each component in the policy. A = array S = set T = tree – = Beyond NUMCOMPS
overrideseclabel	CHAR(1)	Indicates the behavior when a user's security label and exemption credentials do not allow them to insert or update a data row with the security that is label provided on the INSERT or UPDATE SQL statement. • Y: The security label provided is ignored and replaced by the user's security label for write access. • N: Return an error when not authorized to write a security label.

SYSSECPOLICYCOMPONENTS

The **syssecpolicycomponents** system catalog table records the components for each security policies. It has these columns.

Column	Type	Explanation
secpolicyid	INTEGER	Security policy ID
compid	INTEGER	ID of a component of the label security policy
compno	SMALLINT	Position of the security label component as it exists in the security policy, starting with position 1.

SYSSECPOLICYEXEMPTIONS

The **syssecpolicyexemptions** system catalog table records the exemptions that have been given to users. It has these columns.

Column	Type	Explanation
grantee	CHAR(32)	The user who has this exemption

Column	Type	Explanation
secpolicyid	INTEGER	ID of the policy on which the exemption is granted
exemption	CHAR(6)	The exemption given to the user who is identified in the GRANTEE column. The six characters have the following meanings: 1 = Read array 2 = Read set 3 = Read tree 4 = Write array 5 = Write set 6 = Write tree Each character has one of the following values: E = Exempt D = Write down exemption U = Write up exemption – = No exemption

SYSSEQUENCES

The **syssequences** system catalog table lists the sequence objects that exist in the database. The **syssequences** table has the following columns.

Column	Type	Explanation
seqid	SERIAL	Code uniquely identifying the sequence object
tabid	INTEGER	Identifying code of the sequence as a table object
start_val	INT8	Starting value of the sequence
inc_val	INT8	Value of the increment between successive values
max_val	INT8	Largest possible value of the sequence
min_val	INT8	Smallest possible value of the sequence
cycle	CHAR(1)	Zero means NOCYCLE, 1 means CYCLE
restart_val	INT8	Starting value of the sequence after ALTER SEQUENCE RESTART was run
cache	INTEGER	Number of preallocated values in sequence cache
order	CHAR(1)	Zero means NOORDER, 1 means ORDER

SYSSURROGATEAUTH

The **syssurrogateauth** system catalog table stores trusted user and surrogate user information.

The **syssurrogateauth** system catalog table is populated when the GRANT SETSESSIONAUTH statement is run. Users or roles specified in the TO clause are added to **trusteduser** column. Users specified in the ON clause are added to **surrogateuser** column.

For example, consider the following statement:

```
GRANT SETSESSIONAUTH ON bill, john TO mary, peter;
```

Entries in the **syssurrogateauth** table are created as follows:

```
trusteduser    surrogateuser

mary           bill
mary           john
peter          bill
peter          john
```

The **syssurrogateauth** table has the following columns.

Table 60. SYSSURROGATEAUTH table column descriptions

Column	Type	Explanation
trusteduser	CHAR(32)	Trusted user name or role.
surrogateuser	CHAR(32)	Surrogate user name.

SYSSYNONYMS

The **syssynonyms** system catalog table is unused. The **syssyntable** table describes synonyms. The **syssynonyms** system catalog table has the following columns.

Table 61. SYSSYNONYMS table column descriptions

Column	Type	Explanation
owner	VARCHAR(32)	Name of the owner of the synonym
synname	VARCHAR(128)	Name of the synonym
created	DATE	Date when the synonym was created

Table 61. SYSSYNONYMS table column descriptions

(continued)

Column	Type	Explanation
tabid	INTEGER	Identifying code of a table, sequence, or view

SYSSYNTABLE

The **syssyntable** system catalog table outlines the mapping between each public or private synonym and the database object (table, sequence, or view) that it represents. It contains one row for each entry in the **systables** table that has a **tabtype** value of `P` or `S`. The **syssyntable** table has the following columns.

Column	Type	Explanation
tabid	INTEGER	Identifying code of the public synonym
servername	VARCHAR(128)	Name of an external database server
dbname	VARCHAR(128)	Name of an external database
owner	VARCHAR(32)	Name of the owner of an external object
tabname	VARCHAR(128)	Name of an external table or view
btabid	INTEGER	Identifying code of a base table, sequence, or view

ANSI-compliant databases do not support public synonyms; their **syssyntable** tables can describe only synonyms whose **syssyntable.tabtype** value is `P`.

If you define a synonym for an object that is in your current database, only the **tabid** and **btabid** columns are used. If you define a synonym for a table that is external to your current database, the **btabid** column is not used, but the **tabid**, **servername**, **dbname**, **owner**, and **tabname** columns are used.

The **tabid** column maps to **systables.tabid**. With the **tabid** information, you can determine additional facts about the synonym from **systables**.

An index on the **tabid** column allows only unique values. The **btabid** column is indexed to allow duplicate values.

SYSTABAMDATA

The **systabamdata** system catalog table stores the table-specific hashing parameters of tables that were created with a primary access method.

The **systabamdata** table has the following columns.

Table 62. SYSTABAMDATA table column descriptions

Column	Type	Explanation
tabid	INTEGER	Identifying code of the table
am_param	LVARCHAR(8192)	Access method parameter choices
am_space	VARCHAR(128)	Name of the storage space holding the data values

The **am_param** column stores configuration parameters that determine how a primary access method accesses a given table. Each configuration parameter in the **am_param** list has the format *keyword=value* or *keyword*.

The **am_space** column specifies the location of the table. It might be located in a cooked file, a different database, or an sbspace within the database server.

The **tabid** column is the primary key to the **systables** table. This column is indexed and must contain unique values.

SYSTABAUTH

The **systabauth** system catalog table describes each set of privileges that are granted on a table, view, sequence, or synonym. It contains one row for each set of table privileges that are granted in the database; the REVOKE statement can modify a row. The **systabauth** table has the following columns.

Table 63. SYSTABAUTH table column descriptions

Column	Type	Explanation
grantor	VARCHAR(32)	Name of the grantor of privilege
grantee	VARCHAR(32)	Name of the grantee of privilege
tabid	INTEGER	Value from **systables.tabid** for database object
tabauth	CHAR(9) CHAR(8)	Pattern that specifies privileges on the table, view, synonym, or sequence: s *or* S = Select u *or* U = Update ***** = Column-level privilege i *or* I = Insert d *or* D = Delete x *or* X = Index a *or* A = Alter r *or* R = References n *or* N = Under privilege

If the **tabauth** column shows a privilege code in uppercase (for example, `S` for Select), this indicates that the user also has the option to grant that privilege to others. Privilege codes listed in lowercase (for example, `s` for select) indicate that the user has the specified privilege, but cannot grant it to others.

A hyphen (`-`) indicates the absence of the privilege corresponding to that position within the **tabauth** pattern.

A **tabauth** value with an asterisk (*) means column-level privileges exist; see also **syscolauth** (page SYSINDEXES on page 1350). (In , the **Privileges** option of the **Info** command for a specified table can display the column-level privileges on that table.)

A composite index on **tabid**, **grantor**, and **grantee** allows only unique values. A composite index on **tabid** and **grantee** allows duplicate values.

SYSTABLES

The **systables** system catalog table contains a row for each table object (a table, view, synonym, or in HCL Informix®, a sequence) that has been defined in the database, including the tables and views of the system catalog.

Table 64. SYSTABLES table column descriptions

Column	Type	Explanation
tabname	VARCHAR(128)	Name of table, view, synonym, or sequence
owner	CHAR(32)	Owner of table (user **informix** for system catalog tables and *username* for database tables)
partnum	INTEGER	Physical storage location code
tabid	SERIAL	System-assigned sequential identifying number
rowsize	SMALLINT	Maximum row size in bytes (< 32,768)
ncols	SMALLINT	Number of columns in the table
nindexes	SMALLINT	Number of indexes on the table
nrows	FLOAT	Number of rows in the table
created	DATE	Date when table was created or last modified
version	INTEGER	Number that changes when table is altered
tabtype	CHAR(1)	Code indicating the type of table object: • T = Table • E = External Table • V = View • Q = Sequence

Table 64. SYSTABLES table column descriptions (continued)

Column	Type	Explanation
		• P = Private synonym • S = Public synonym (Type S is unavailable in an ANSI-compliant database.)
locklevel	CHAR(1)	Lock mode for the table: • B = Page and row level • P = Page level • R = Row level
npused	FLOAT	Number of data pages that have ever been initialized in the tablespace by the database server
fextsize	INTEGER	Size of initial extent (in KB)
nextsize	INTEGER	Size of all subsequent extents (in KB)
flags	SMALLINT	Codes for classifying permanent tables: **ROWID** 1 - Has rowid column defined **UNDER** 2 - Table created under a supertable **VIEWREMOTE** 4 - View is based on a remote table **CDR** 8 - Has CDRCOLS defined **RAW** 16 - (Informix®) RAW table **EXTERNAL** 32- External table **AUDIT** 64 - Audit table attribute - FGA **AQT** 128 - View is an AQT for DWA offloading

Table 64. SYSTABLES table column descriptions (continued)

Column	Type	Explanation
		VIRTAQT 256 - View is a virtual AQT
site	VARCHAR(128)	Reserved for future use
dbname	VARCHAR(128)	Reserved for future use
type_xid	INTEGER	Code from **sysxtdtypes.extended_id** for typed tables, or `0` for untyped tables
am_id	INTEGER	Access method code (key to **sysams** table) NULL or 0 indicates built-in storage manager
pagesize	INTEGER	The pagesize, in bytes, of the dbspace (or dbspaces, if the table is fragmented) where the table data resides.
ustlowts	DATETIME YEAR TO FRACTION (5)	When table, row, and page-count statistics were last recorded
secpolicyid	INTEGER	ID of the SECURITY policy attached to the table. NULL for non-protected tables
protgranularity	CHAR(1)	LBAC granularity level: • R: Row level granularity • C: Column level granularity • B: Both column and row granularity • Blank for non-protected tables
statlevel	CHAR(1)	Statistics level • T = table • F = fragment • A = automatic
statchange	SMALLINT	For internal use only

Each table, view, sequence, and synonym recorded in the **systables** table is assigned a **tabid**, which is a system-assigned SERIAL value that uniquely identifies the object. The first 99 **tabid** values are reserved for the system catalog. The **tabid** of the first user-defined table object in a database is always 100.

The **tabid** column is indexed and contains only unique values. A composite index on the **tabname** and **owner** columns also requires unique values.

The version column contains an encoded number that is stored in **systables** when a new table is created. Portions of this value are incremented when data-definition statements, such as ALTER INDEX, ALTER TABLE, DROP INDEX, and CREATE INDEX, are performed on the table.

In the **flags** column, ST_RAW represents a nonlogging permanent table in a database that supports transaction logging.

The setting of the SQL_LOGICAL_CHAR parameter is encoded into the **systables.flags** column value in the row that describes the ' **VERSION**' table. Note the leading blank space in the identifier of this system-generated table.

To determine whether the database enables the SQL_LOGICAL_CHAR configuration parameter, which can apply logical character semantics to the declarations of character columns, you can execute the following query:

```
SELECT flags INTO $value FROM 'informix'.systables WHERE tabname = ' VERSION';
```

Because the SQL_LOGICAL_CHAR setting is encoded in the two least significant bits of the " **VERSION.flags**" value, you can calculate its setting from the returned **flags** value by the following formula:

```
SQL_LOGICAL_CHAR = (value & 0x03) + 1
```

Here `&` is the bitwise `AND` operator. Any SQL_LOGICAL_CHAR setting greater than 1 indicates that SQL_LOGICAL_CHAR was enabled when the database was created, and that explicit or default maximum size specifications of character columns are multiplied by that setting.

When a prepared statement that references a database table is executed, the version value is checked to make sure that nothing has changed since the statement was prepared. If the version value has been changed by DDL operations that modified the table schema while automatic recompilation was disabled by the IFX_AUTO_REPREPARE setting of the SET ENVIRONMENT statement, the prepared statement is not executed, and you must prepare the statement again.

The **npused** column does not reflect the number of pages used for BYTE or TEXT data, nor the number of pages that are freed in DELETE or TRUNCATE operations.

The **nrows** column and the **npused** columns might not accurately reflect the number of rows and the number of data pages used by an external table unless the NUMROWS clause was specified when the external table was created. See the *HCL® Informix® Administrator's Guide* for more information.

The **systables** table has two rows that store information about the database locale: GL_COLLATE with a **tabid** of `90` and GL_CTYPE with a **tabid** of `91`. To view these rows, enter the following SELECT statement:

```
SELECT * FROM systables WHERE tabid=90 OR tabid=91;
```

SYSTRACECLASSES

The **systraceclasses** system catalog table contains the names and identifiers of trace classes. The **systraceclasses** table has the following columns.

Table 65. SYSTRACECLASSES table column descriptions

Column	Type	Explanation
name	CHAR(18)	Name of the class of trace messages

Table 65. SYSTRACECLASSES table column descriptions (continued)

Column	Type	Explanation
classid	SERIAL	Identifying code of the trace class

A *trace class* is a category of trace messages that you can use in the development and testing of new modules and user-defined routines. Developers use the tracing facility by calling the appropriate routines within their code.

To create a new trace class, insert a row directly into the **systraceclasses** table. By default, all users can view this table, but only users with the DBA privilege can modify it.

The database cannot support tracing unless the MITRACE_OFF configuration parameter is undefined.

A unique index on the **name** column requires each trace class to have a unique name. The database server assigns to each class a unique sequential code. The index on this **classid** column also allows only unique values.

SYSTRACEMSGS

The **systracemsgs** system catalog table stores internationalized trace messages that you can use in debugging user-defined routines.

The **systracemsgs** table has the following columns.

Table 66. SYSTRACEMSGS table column descriptions

Column	Type	Explanation
name	VARCHAR(128)	Name of the message
msgid	SERIAL	Identifying code of the message template
locale	CHAR(36)	Locale with which this version of the message is associated (for example, **en_us.8859-1**)
seqno	SMALLINT	Reserved for future use
message	VARCHAR(255)	The message text

module developers create a trace message by inserting a row directly into the **systracemsgs** table. After a message is created, the development team can specify it either by name or by **msgid** code, using trace statements that the provides.

To create a trace message, you must specify its name, locale, and text. By default, all users can view the **systracemsgs** table, but only users with the DBA privilege can modify it.

The database cannot support tracing unless the MITRACE_OFF configuration parameter is undefined.

A unique composite index is defined on the **name** and **locale** columns. Another unique index is defined on the **msgid** column.

SYSTRIGBODY

The **systrigbody** system catalog table contains the ASCII text of the trigger definition and the linearized code for the trigger. *Linearized code* is binary data and code that is represented in ASCII format.

Important: The database server uses the linearized code that is stored in **systrigbody**. You must not alter the content of rows that contain linearized code.

The **systrigbody** table has the following columns.

Table 67. SYSTRIGBODY table column descriptions

Column	Type	Explanation
trigid	INTEGER	Identifying code of the trigger
datakey	CHAR(1)	Code specifying the type of data: A = ASCII text for the body, triggered actions B = Linearized code for the body D = English text for the header, trigger definition H = Linearized code for the header S = Linearized code for the symbol table
seqno	INTEGER	Page number of this data segment
data	CHAR(256)	English text or linearized code

A composite index on the **trigid**, **datakey**, and **seqno** columns allows only unique values.

SYSTRIGGERS

The **systriggers** system catalog table contains information about the SQL triggers in the database. This information includes the triggering event and the correlated reference specification for the trigger. The **systriggers** table has the following columns.

Table 68. SYSTRIGGERS table column descriptions

Column	Type	Explanation
trigid	SERIAL	Identifying code of the trigger
trigname	VARCHAR(128)	Name of the trigger
owner	VARCHAR(32)	Name of the owner of the trigger
tabid	INTEGER	Identifying code of the triggering table
event	CHAR(1)	Code for the type of triggering event:

Table 68. SYSTRIGGERS table column descriptions (continued)

Column	Type	Explanation
		D = Delete trigger I = Insert trigger U = Update trigger S = Select trigger d = INSTEAD OF Delete trigger i = INSTEAD OF Insert trigger u = INSTEAD OF Update trigger
old	VARCHAR(128)	Name of value before update
new	VARCHAR(128)	Name of value after update
mode	CHAR(1)	Reserved for future use
collation	CHAR(32)	Collating order at the time when the routine was created

A composite index on the **trigname** and **owner** columns allows only unique values. An index on the **trigid** column also requires unique values. An index on the **tabid** column allows duplicate values.

SYSUSERS

The **sysusers** system catalog table lists the authorization identifier of every individual user, or public for the PUBLIC group, who holds database-level access privileges. This table also lists the name of every role that holds access privileges on any object in the database.

This system catalog table has the following columns:

Table 69. SYSUSERS table column descriptions

Column	Type	Explanation
username	VARCHAR(32)	Name of the database user or role. An index on **username** allows only unique values. The **username** value can be the login name of a user or the name of a role.
usertype	CHAR(1)	Code specifying the highest database-level privilege held by **username**, where **username** is an individual user or the PUBLIC group, or a role name. The valid codes are: D = DBA (all privileges) R = Resource (create UDRs, UDTs, permanent tables, and indexes)

Table 69. SYSUSERS table column descriptions

(continued)

Column	Type	Explanation
		C = Connect (work with existing tables) G = Role U = Default role. When a user is assigned a default role, an implicit connection to the database is granted to the user. This is the role the user has before being granted a C, D, or R role.
priority	SMALLINT	Reserved for future use.
password	CHAR(16)	Reserved for future use.
defrole	VARCHAR(32)	Name of the default role.

SYSVIEWS

The **sysviews** system catalog table describes each view in the database. Because it stores the SELECT statement that created the view, **sysviews** can contain multiple rows for each view. It has the following columns.

Column	Type	Explanation
tabid	INTEGER	Identifying code of the view
seqno	SMALLINT	Line number of the SELECT statement
viewtext	CHAR(256)	Actual SELECT statement used to create the view

A composite index on **tabid** and **seqno** allows only unique values.

SYSVIOLATIONS

The **sysviolations** system catalog table stores information about constraint violations for base tables.

This table is updated when the DELETE, INSERT, MERGE, or UPDATE statement detects a violation of an enabled constraint or unique index in a database table for which the START VIOLATIONS TABLE statement of SQL has created an associated violations table (and for Informix®, a diagnostics table). For each base table that has an active violations table, the **sysviolations** table has a corresponding row, with the following columns.

Column	Type	Explanation
targettid	INTEGER	Identifying code of the *target table* (the base table on which the violations table and the diagnostic table are defined)

Column	Type	Explanation
viotid	INTEGER	Identifying code of the violations table
diatid	INTEGER	Identifying code of the diagnostics table
maxrows	INTEGER	Maximum number of rows that can be inserted into the diagnostics table by a single insert, update, or delete operation on a target table that has a filtering mode object defined on it.

The **maxrows** column also signifies the maximum number of rows that can be inserted in the diagnostics table during a single operation that enables a disabled object or that sets a disabled object to filtering mode (provided that a diagnostics table exists for the target table). If no maximum is specified for the diagnostics or violations table, then **maxrows** contains a NULL value.

The primary key of this table is the **targettid** column. An additional unique index is also defined on the **viotid** column.

HCL Informix® also has a unique index on the **diatid** column.

SYSXADATASOURCES

The **sysxadatasources** system catalog table stores XA data sources.

The **sysxadatasources** table has the following columns.

Column	Type	Explanation
xa_datasrc_owner	CHAR(32)	The user ID of the XA data source owner
xa_datasrc_name	VARCHAR(128)	The name of the XA data source
xa_datasrc_rmid	SERIAL	Unique RMID of the XA data source
xa_source_typeid	INTEGER	XA data source type ID

SYSXASOURCETYPES

The **sysxasourcetypes** system catalog table stores XA data source types.

The **sysxasourcetypes** table has the following columns.

Column	Type	Explanation
xa_source_typeid	SERIAL	A unique identifier for the source type
xa_source_owner	CHAR(32)	The user ID of the owner
xa_source_name	VARCHAR(128)	The name of the source type
xa_flags	INTEGER	
xa_version	INTEGER	

Column	Type	Explanation
xa_open	INTEGER	UDR ID of xa_open_entry
xa_close	INTEGER	UDR ID of xa_close_entry
xa_start	INTEGER	UDR ID of xa_start entry
xa_end	INTEGER	UDR ID of xa_end_entry
xa_rollback	INTEGER	UDR ID of xa_rollback_entry
xa_prepare	INTEGER	UDR ID of xa_prepare_entry
xa_commit	INTEGER	UDR ID of xa_commit_entry
xa_recover	INTEGER	UDR ID of xa_recover_entry
xa_forget	INTEGER	UDR ID of xa_forget_entry
xa_complete	INTEGER	UDR ID of xa_complete_entry

SYSXTDDESC

The **sysxtddesc** system catalog table provides a text description of each user-defined data type (UDT) defined in the database. The **sysxtddesc** table has the following columns.

Column	Type	Explanation
extended_id	INTEGER	Code uniquely identifying the extended data types
seqno	SMALLINT	Value to order and identify one line of the description of the UDT A new line is created only if the remaining text string is larger than 255 bytes.
description	CHAR(256)	Textual description of the extended data type

A composite index on **extended_id** and **seqno** allows duplicate values.

SYSXTDTYPEAUTH

The **sysxtdtypeauth** system catalog table identifies the privileges on each UDT (user-defined data type).

The **sysxtdtypeauth** table contains one row for each set of privileges granted and has the following columns:

Column	Type	Explanation
grantor	VARCHAR(32)	Name of grantor of privilege
grantee	VARCHAR(32)	Name of grantee of privilege
type	INTEGER	Code identifying the UDT
auth	CHAR(2)	Code identifying privileges on the UDT:

Column	Type	Explanation
		n *or* N = Under privilege u *or* U = Usage privilege

If the privilege code in the **auth** column is upper case (for example, 'U' for usage), a user who has this privilege can also grant it to others. If the code is in lower case, a user who has the privilege cannot grant it to others.

A composite index on **type**, **grantor**, and **grantee** allows only unique values. A composite index on the **type** and **grantee** columns allows duplicate values.

SYSXTDTYPES

The **sysxtdtypes** system catalog table has an entry for each UDT (user-defined data type), including opaque and distinct data types and complex data types (named ROW types, unnamed ROW types, and COLLECTION types), that is defined in the database.

The **sysxtdtypes** table has the following columns.

Table 70. SYSXTDTYPES table column descriptions

Column	Type	Explanation
extended_id	SERIAL	Unique identifying code for extended data type
domain	CHAR(1)	Code for the domain of the UDT
mode	CHAR(1)	Code classifying the UDT: • B = Base (opaque) type • C = Collection type or unnamed ROW type • D = Distinct type • R = Named ROW type • S = Reserved for internal use • T = Reserved for internal use • ' ' (blank) = Built-in type
owner	VARCHAR(32)	Name of the owner of the UDT
name	VARCHAR(128)	Name of the UDT
type	SMALLINT	Code classifying the UDT
source	INTEGER	The **sysxtdtypes** reference (for distinct types only)

Table 70. SYSXTDTYPES table column descriptions

(continued)

Column	Type	Explanation
		Zero (0) indicates that a distinct UDT was created from a built-in data type.
maxlen	INTEGER	The maximum length for variable-length data types Zero indicates a fixed-length UDT.
length	INTEGER	The length in bytes for fixed-length data types Zero indicates a variable-length UDT.
byvalue	CHAR(1)	'T' = UDT is passed by value 'F' = UDT is not passed by value
cannothash	CHAR(1)	'T' = UDT is hashable by default hash function 'F' = UDT is not hashable by default function
align	SMALLINT	Alignment (= 1, 2, 4, *or* 8) for this UDT
locator	INTEGER	Locator key for unnamed ROW type

Each extended data type is characterized by a unique identifier, called an extended identifier (**extended_id**), a data type identifier (**type**), and the length and description of the data type.

For distinct types created from built-in data types, the **type** column codes correspond to the value of the **syscolumns.coltype** column (indicating the source type) as listed on page SYSCOLUMNS on page 1336, but incremented by the hexadecimal value 0x0000800. The file `$INFORMIXDIR/incl/esql/sqltypes.h` contains information about **sysxtdtypes.type** and **syscolumns.coltype** codes.

An index on the **extended_id** column allows only unique values. An index on the **locator** column allows duplicate values, as does a composite index on the **name** and **owner** columns. A composite index on the **type** and **source** columns also allows duplicate values.

Information Schema

The Information Schema consists of read-only views that provide information about all the tables, views, and columns in the current database server to which you have access. These views also provide information about SQL dialects (such as HCL Informix®, Oracle, or Sybase) and SQL standards. Note that unlike a system catalog, whose tables describes an individual database, these views describe the HCL Informix® instance, rather than a single database.

This version of the Information Schema views is an X/Open CAE standard. These standards are provided so that applications developed on other database systems can obtain HCL Informix® system catalog information without accessing the HCL Informix® system catalog tables directly.

Important: Because the X/Open CAE standard for Information Schema views differs from ANSI-compliant Information Schema views, it is recommended that you do not install the X/Open CAE Information Schema views on ANSI-compliant databases.

The following Information Schema views are available:

- **tables**
- **columns**
- **sql_languages**
- **server_info**

Sections that follow contain information about how to generate and access Information Schema views and information about their structure.

Generating the Information Schema Views

About this task

The Information Schema views are generated automatically when you, as DBA, run the following command:

```
dbaccess database-name $INFORMIXDIR/etc/xpg4_is.sql
```

The views display data from the system catalog tables. If tables, views, or routines exist with any of the same names as the Information Schema views, you must either rename those database objects or rename the views in the script before you can install the views. You can drop the views with the DROP VIEW statement on each view. To re-create the views, rerun the script.

Important: In addition to the columns specified for each Information Schema view, individual vendors might include additional columns or change the order of the columns. It is recommended that applications not use the forms SELECT * or SELECT table-name* to access an Information Schema view.

Accessing the Information Schema Views

All Information Schema views have the Select privilege granted to PUBLIC WITH GRANT OPTION so that all users can query the views. Because no other privileges are granted on the Information Schema views, they cannot be updated.

You can query the Information Schema views as you would query any other table or view in the database.

Structure of the Information Schema Views

The following Information Schema views are described in this section:

- **tables**
- **columns**
- **sql_languages**
- **server_info**

In order to accept long identifier names, most of the columns in the views are defined as VARCHAR data types with large maximum sizes.

The tables Information Schema View

The **tables** Information Schema view contains one row for each table to which you have access. It contains the following columns.

Column	Data Type	Explanation
table_schema	VARCHAR(32)	Name of owner of table
table_name	VARCHAR(128)	Name of table or view
table_type	VARCHAR(128)	BASE TABLE for table or VIEW for view
remarks	VARCHAR(255)	Reserved for future use

The visible rows in the **tables** view depend on your privileges. For example, if you have one or more privileges on a table (such as Insert, Delete, Select, References, Alter, Index, or Update on one or more columns), or if privileges are granted to PUBLIC, you see the row that describes that table.

The columns Information Schema View

The **columns** Information Schema view contains one row for each accessible column. It contains the following columns.

Table 71. Description of the columns Information Schema View

Column	Data Type	Explanation
table_schema	VARCHAR(128)	Name of owner of table
table_name	VARCHAR(128)	Name of table or view
column_name	VARCHAR(128)	Name of the column in the table or view
ordinal_position	INTEGER	Position of the column within its table The **ordinal_position** value is a sequential number that starts at 1 for the first column. This is the HCL Informix® extension to XPG4.
data_type	VARCHAR(254)	Name of the data type of the column, such as CHARACTER or DECIMAL

Table 71. Description of the columns Information Schema View (continued)

Column	Data Type	Explanation
char_max_length	INTEGER	Maximum length (in bytes) for character data types; NULL otherwise
numeric_precision	INTEGER	Uses one of the following values: • Total number of digits for exact numeric data types (DECIMAL, INTEGER, MONEY, SMALLINT) • Number of digits of mantissa precision (machine-dependent) for approximate data types (FLOAT, SMALLFLOAT) • NULL for all other data types.
numeric_prec_radix	INTEGER	Uses one of the following values: • 2 = Approximate data types (FLOAT and SMALLFLOAT) • 10 = Exact numeric data types (DECIMAL, INTEGER, MONEY, and SMALLINT) • NULL for all other data types
numeric_scale	INTEGER	Number of significant digits to the right of the decimal point for DECIMAL and MONEY data types 0 for INTEGER and SMALLINT types NULL for all other data types
datetime_precision	INTEGER	Number of digits in the fractional part of the seconds for DATE and DATETIME columns; NULL otherwise This column is the HCL Informix® extension to XPG4.
is_nullable	VARCHAR(3)	Indicates whether a column allows NULL values; either YES or NO
remarks	VARCHAR(254)	Reserved for future use

The sql_languages Information Schema View

The **sql_languages** Information Schema view contains a row for each instance of conformance to standards that the current database server supports. The **sql_languages** view contains the following columns.

Column	Data Type	Explanation
source	VARCHAR(254)	Organization defining this SQL version
source_year	VARCHAR(254)	Year the source document was approved
conformance	VARCHAR(254)	Standard to which the server conforms
integrity	VARCHAR(254)	Indication of whether this is an integrity enhancement feature; either `YES` or `NO`
implementation	VARCHAR(254)	Identification of the SQL product of the vendor
binding_style	VARCHAR(254)	Direct, module, or other binding style
programming_lang	VARCHAR(254)	Host language for which binding style is adapted

The **sql_languages** view is completely visible to all users.

The server_info Information Schema View

The **server_info** Information Schema view describes the database server to which the application is currently connected. It contains two columns.

Column	Data Type	Explanation
server_attribute	VARCHAR(254)	An attribute of the database server
attribute_value	VARCHAR(254)	Value of the **server_attribute** as it applies to the current database server

Each row in this view provides information about one attribute. X/Open-compliant databases must provide applications with certain required information about the database server.

The **server_info** view includes the following **server_attribute** information.

server_attribute	Explanation
identifier_length	Maximum number of bytes for a user-defined identifier
row_length	Maximum number of bytes in a row
userid_length	Maximum number of bytes in a user name
txn_isolation	Initial transaction isolation level for the database server: `Read Uncommitted` (= Default isolation level for databases with no transaction logging; also called `Dirty Read`) `Read Committed` (= Default isolation level for databases that are not ANSI-compliant, but that support explicit transaction logging)

server_attribute	Explanation
	`Serializable` (= Default isolation level for ANSI-compliant databases; also called `Repeatable Read`)
collation_seq	Assumed ordering of the character set for the database server The following values are possible: `ISO 8859-1 EBCDIC` The default HCL Informix® representation shows ISO 8859-1.

The **server_info** view is completely visible to all users.

Data types

Every column in a table in a database is assigned a data type. The data type precisely defines the kinds of values that you can store in that column.

These topics describe built-in and extended data types, casting between two data types, and operator precedence.

Summary of data types

HCL Informix® supports the most common set of built-in data types. Additionally, an extended set of data types are supported on the database server.

You can use both *built-in* data types (which are system-defined) and *extended* data types (which you can define) in the following ways:

- Use them to create columns within database tables.
- Declare them as arguments and as returned types of routines.
- Use them as base types from which to create DISTINCT data types.
- Cast them to other data types.
- Declare and access host variables of these types in SPL and ESQL/C.

You assign data types to columns with the CREATE TABLE statement and change them with the ALTER TABLE statement. When you change an existing column data type, all data is converted to the new data type, if possible.

For information about the ALTER TABLE and CREATE TABLE statements, on SQL statements that create specific data types, that create and drop casts, and on other data type topics, see the *HCL® Informix® Guide to SQL: Syntax*.

For information about how to create and use complex data types supported by HCL Informix®, see the *IBM® Informix® Database Design and Implementation Guide*. For information about how to create user-defined data types, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

Some data types can be used in distributed SQL operations, while others can be used only in SQL operations within the same database.

Built-in data types supported in local and distributed SQL operations

The following table lists all of the built-in SQL data types that Informix® supports. These built-in SQL data types are valid in all Informix® SQL transactions, including data-manipulation language (DML) operations of these types:

- Operations on objects in the local database
- Cross-database operations on objects in databases of the local server instance
- Cross-server operations on objects in databases of two or more database server instances

Table 72. Data types supported in all operations

Data type	Explanation
BIGINT data type on page 1394	Stores 8-byte integer values from $-(2^{63}-1)$ to $2^{63}-1$
BIGSERIAL data type on page 1394	Stores sequential, 8-byte integers from 1 to $2^{63}-1$
BSON and JSON built-in opaque data types on page	The BSON data type is the binary representation of a JSON data type format for serializing JSON documents. The JSON data type is a plain text format for entering and displaying structured data.
BYTE data type on page 1396	Stores any kind of binary data, up to 2^{31} bytes in length
#unique_1911	Stores character strings; collation is in code-set order
CHARACTER(n) data type on page 1397	Is a synonym for CHAR
CHARACTER VARYING(m,r) data type on page 1397	Stores character strings of varying length (ANSI-compliant); collation is in code-set order
DATE data type on page 1399	Stores calendar dates
#unique_1915	Stores calendar date combined with time of day
DEC data type on page 1399	Is a synonym for DECIMAL
DECIMAL on page 1399	Stores floating-point numbers with definable precision; if database is ANSI-compliant, the scale is zero
DECIMAL (p,s) Fixed Point on page 1400	Stores fixed-point numbers of defined scale and precision
DOUBLE PRECISION data types on page 1402	Synonym for FLOAT
FLOAT(n) on page 1402	Stores double-precision floating-point numbers corresponding to the **double** data type in C
INT data type on page 1403	Is a synonym for INTEGER
INT8 on page 1403	Stores 8-byte integer values from $-(2^{63}-1)$ to $2^{63}-1$

Table 72. Data types supported in all operations (continued)

Data type	Explanation
INTEGER data type on page 1403	Stores whole numbers from -2,147,483,647 to +2,147,483,647
#unique_1924	Stores a span of time (or level of effort) in units of *years* and *months*.
#unique_1924	Stores a span of time in a contiguous set of units of *days*, *hours*, *minutes*, *seconds*, and *fractions of a second*
MONEY(p,s) data type on page 1406	Stores currency amounts
NCHAR(n) data type on page 1407	Same as CHAR, but can support localized collation
NUMERIC(p,s) data type on page 1408	Synonym for DECIMAL(*p*,*s*)
NVARCHAR(m,r) data type on page 1408	Same as VARCHAR, but can support localized collation
REAL data type on page 1409	Is a synonym for SMALLFLOAT
SERIAL(n) data type on page 1412	Stores sequential integers (> 0) in positive range of INT
SERIAL8(n) data type on page 1412	Stores sequential integers (> 0) in positive range of INT8
SMALLFLOAT on page 1415	Stores single-precision floating-point numbers corresponding to the **float** data type of the C language
SMALLINT data type on page 1415	Stores whole numbers from -32,767 to +32,767
TEXT data type on page 1415	Stores any kind of text data, up to 2^{31} bytes in length
VARCHAR(m,r) data type on page 1417	Stores character strings of varying length (up to 255 bytes); collation is in code-set order

In cross-server MERGE operations, the source table (but not the target table) can be in a database of a remote Informix® server.

For the character data types (CHAR, CHAR VARYING, LVARCHAR, NCHAR, NVARCHAR, and VARCHAR), a data string can include letters, digits, punctuation, whitespace, diacritical marks, ligatures, and other printable symbols from the code set of the database locale. For **UTF-8** and for code sets of some East Asian locales, multibyte characters are supported within data strings.

Built-in data types supported only in local database SQL operations

The following table lists the data types that Informix® supports only for use in SQL operations in a local database.

Table 73. Data types supported in a local database

Data type	Explanation
BLOB data type on page 1395	Stores binary data in random-access chunks
The binary18 data type on page	Stores 18 byte binary-encoded strings
The binaryvar data type on page	Stores binary-encoded strings with a maximum length of 255 bytes
BOOLEAN data type on page 1396	Stores Boolean values true and false
CLOB data type on page 1398	Stores text data in random-access chunks
DISTINCT data types on page 1401	Stores data in a user-defined type that has the same format as a source type on which it is based, but its casts and functions can differ from those on the source type
Calendar data type on page	Stores a calendar for a TimeSeries data type
CalendarPattern data type on page	Stores the structure of the calendar pattern for a Calendar data type
IDSSECURITYLABEL data type on page 1403	Stores LBAC security label objects.
LIST(e) data type on page 1404	Stores a sequentially ordered collection of elements, all of the same data type, *e*; allows duplicate values
The lld_locator data type on page	Stores a large object identifier
The lld_lob data type on page	Stores the location of a smart large object and specifies whether the object contains binary or character data
LVARCHAR(m) data type on page 1405	Stores variable-length strings of up to 32,739 bytes
MULTISET(e) data type on page 1406	Stores a non-ordered collection of values, with elements all of the same data type, *e*; allows duplicate values.
The node data type for querying hierarchical data on page	Stores a combination of integers and decimal points that represents hierarchical relationships, of variable length up to 256 characters
OPAQUE data types on page 1408	Stores a user-defined data type whose internal structure is inaccessible to the database server

Table 73. Data types supported in a local database (continued)

Data type	Explanation
ROW data type, Named on page 1409	Stores a named ROW type
ROW data type, Unnamed on page 1410	Stores an unnamed ROW type
SET(e) data type on page 1414	Stores a non-ordered collection of elements, all of the same data type, *e*; does not allow duplicate values
TimeSeries data type on page	Stores a collection of row subtypes

These extended data types of Informix® are individually described in other topics. These data types are valid in local operations on databases where the data types are defined.

Extended data types in cross-database distributed SQL transactions

Distributed operations on other databases of the same Informix® instance can access BOOLEAN, BLOB, CLOB, and LVARCHAR data types, which are implemented as built-in opaque types. Such operations can also access DISTINCT types whose base types are built-in types, and user-defined types (UDTs), if the UDTs and DISTINCT types are explicitly cast to built-in types, and if all of the UDTs, casts, and DISTINCT types are defined in all the participating databases.

You cannot, however, reference the following extended data types in cross-database transactions that access multiple databases of the local Informix® instance:

- UDTs that are not cast to built-in data types
- DISTINCT types that are not cast to built-in data types
- Collection data types
- Named or unnamed ROW data types

Extended data types in cross-server distributed SQL transactions

Distributed SQL transactions and function calls that access databases of other Informix® instances cannot return values of complex or smart large object data types, nor of most distinct or built-in opaque data types. Among the extended data types, only the following can be accessed in cross-server SQL operations:

- Any non-opaque built-in data type
- BOOLEAN
- DISTINCT of non-opaque built-in types
- DISTINCT of BOOLEAN
- DISTINCT of LVARCHAR
- DISTINCT of any of the DISTINCT types listed above

- IDSSECURITYLABEL
- LVARCHAR

A cross-server distributed SQL transaction can support DISTINCT data types only if they are cast explicitly to built-in types, and all of the DISTINCT types, their data type hierarchies, and their casts are defined exactly the same way in each database that participates in the distributed operation. For queries or other DML operations in cross-server UDRs that use the data types in the preceding list as parameters or as returned data types, the UDR must also have the same definition in every participating database.

The built-in DISTINCT data type IDSSECURITYLABEL, which stores security label objects, can be accessed in cross-server and cross-database operations on protected data by users who hold sufficient security credentials. Like local operations on protected data, distributed queries that access remote tables protected by a security policy can return only the qualifying rows that IDSLBACRULES allow, after the database server has compared the security label that secures the data with the security credentials of the user who issues the query.

Related information

Built-In Data Types on page 1419

Extended Data Types on page 1429

Description of Data Types

This section describes the data types that HCL Informix® supports.

BIGINT data type

The BIGINT data type stores integers from $-(2^{63}-1)$ to $2^{63}-1$, which is –9,223,372,036,854,775,807 to 9,223,372,036,854,775,807, in eight bytes.

This data type has storage advantages over INT8 and advantages for some arithmetic operations and sort comparisons over INT8 and DECIMAL data types.

BIGSERIAL data type

The BIGSERIAL data type stores a sequential integer, of the BIGINT data type, that is assigned automatically by the database server when a new row is inserted. The behavior of the BIGSERIAL data type is similar to the SERIAL data type, but with a larger range.

The default BIGSERIAL starting number is 1, but you can assign an initial value, *n*, when you create or alter the table. The value of *n* must be a positive integer in the range of 1 to 9,223,372,036,854,775,807. If you insert the value zero (0) in a BIGSERIAL column, the value that is used is the maximum positive value that already exists in the BIGSERIAL column + 1. If you insert any value that is not zero, that value will be inserted as it is.

A table can have no more than one SERIAL column, but it can have a SERIAL column and either a SERIAL8 column or a BIGSERIAL column.

For information about:

- The SERIAL data type, see SERIAL(n) data type on page 1412
- Using the SERIAL8 data type with the INT8 or BIGINT data type, see Using SERIAL8 and BIGSERIAL with INT8 or BIGINT on page 1395

Using SERIAL8 and BIGSERIAL with INT8 or BIGINT

All the arithmetic operators that are valid for INT8 and BIGINT (such as +, -, *, and /) and all the SQL functions that are valid for INT8 and BIGINT (such as ABS, MOD, POW, and so on) are also valid for SERIAL8 and BIGSERIAL values.

Data conversion rules that apply to INT8 and BIGINT also apply to SERIAL8 and BIGSERIAL, but with a NOT NULL constraint on SERIAL8 or BIGSERIAL.

The value of a SERIAL8 or BIGSERIAL column of one table can be stored in INT8 or BIGINT columns of another table. In the second table, however, the INT8 or BIGINT values are not subject to the constraints on the original SERIAL8 or BIGSERIAL column.

BLOB data type

The BLOB data type stores any kind of binary data in random-access chunks, called sbspaces. Binary data typically consists of saved spreadsheets, program-load modules, digitized voice patterns, and so on. The database server performs no interpretation of the contents of a BLOB column.

A BLOB column can be up to 4 terabytes ($4*2^{40}$ bytes) in length, though your system resources might impose a lower practical limit. The minimum amount of disk space allocated for smart large object data types is 512 bytes.

The term *smart large object* refers to BLOB and CLOB data types. Use CLOB data types (see page CLOB data type on page 1398) for random access to text data. For general information about BLOB and CLOB data types, see Smart large objects on page 1423.

You can use these SQL functions to perform operations on a BLOB column:

- **FILETOBLOB** copies a file into a BLOB column.
- **LOTOFILE** copies a BLOB (or CLOB) value into an operating-system file.
- **LOCOPY** copies an existing smart large object to a new smart large object.

For more information about these SQL functions, see the *HCL® Informix® Guide to SQL: Syntax*.

Within SQL, you are limited to the equality (=) comparison operation and the encryption and decryption functions for BLOB data. (The encryption and decryption functions are described in the *HCL® Informix® Guide to SQL: Syntax*.) To perform additional operations, you must use one of the application programming interfaces (APIs) from within your client application.

You can insert data into BLOB columns in the following ways:

- With the dbload or onload utilities
- With the LOAD statement (DB-Access)

- With the FILETOBLOB function
- From BLOB (**ifx_lo_t**) host variables ()

If you select a BLOB column using DB-Access, only the string `<SBlob value>` is returned; no actual value is displayed.

Related information

FILETOBLOB and FILETOCLOB Functions on page

LOTOFILE Function on page

LOCOPY Function on page

BOOLEAN data type

The BOOLEAN data type stores `TRUE` or `FALSE` data values as a single byte.

The following table shows internal and literal representations of the BOOLEAN data type.

Logical Value	Internal Representation	Literal Representation
TRUE	\0	't'
FALSE	\1	'f'
NULL	Internal Use Only	NULL

You can compare two BOOLEAN values to test for equality or inequality. You can also compare a BOOLEAN value to the Boolean literals '`t`' and '`f`'. BOOLEAN values are not case-sensitive; '`t`' is equivalent to '`T`' and '`f`' to '`F`'.

You can use a BOOLEAN column to store what a Boolean expression returns. In the following example, the value of **boolean_column** is '`t`' if **column1** is less than **column2**, '`f`' if **column1** is greater than or equal to **column2**, and NULL if the value of either **column1** or **column2** is unknown:

```
UPDATE my_table SET boolean_column = lessthan(column1, column2)
```

BYTE data type

The BYTE data type stores any kind of binary data in an undifferentiated byte stream. Binary data typically consists of digitized information, such as spreadsheets, program load modules, digitized voice patterns, and so on.

The term *simple large object* refers to an instance of a TEXT or BYTE data type. No more than 195 columns of the same table can be declared as BYTE and TEXT data types.

The BYTE data type has no maximum size. A BYTE column has a theoretical limit of 2^{31} bytes and a practical limit that your disk capacity determines.

You cannot use the MEDIUM or HIGH options of the UPDATE STATISTICS statement to calculate distribution statistics on BYTE columns.

BYTE objects in DML operations

You can store, retrieve, update, or delete the contents of a BYTE column. You cannot, however, use BYTE operands in arithmetic or string operations, nor assign literals to BYTE columns with the SET clause of the UPDATE statement. You also cannot use BYTE objects in any of the following contexts in a SELECT statement:

- With aggregate functions
- With the IN clause
- With the MATCHES or LIKE clauses
- With the GROUP BY clause
- With the ORDER BY clause

BYTE operands are valid in Boolean expressions only when you are testing for NULL values with the IS NULL or IS NOT NULL operators.

You can use the following methods, which can load rows or update fields, to insert BYTE data:

- With the **dbload** or **onload** utilities
- With the LOAD statement ()
- From BYTE host variables ( )

You cannot use a quoted text string, number, or any other actual value to insert or update BYTE columns.

When you select a BYTE column, you can receive all or part of it. To retrieve it all, use the regular syntax for selecting a column. You can also select any part of a BYTE column by using subscripts, as the next example, which reads the first 75 bytes of the **cat_picture** column associated with the catalog number 10001:

```
SELECT cat_picture [1,75] FROM catalog WHERE catalog_num = 10001
```

A built-in cast converts BYTE values to BLOB values. For more information, see the *IBM® Informix® Database Design and Implementation Guide*.

If you select a BYTE column using the Interactive Schema Editor, only the string "<`BYTE value`>" is returned; no data value is displayed.

Important: If you try to return a BYTE column from a subquery, an error results, even if the column is not used in a Boolean expression nor with an aggregate.

CHARACTER(n) data type

The CHARACTER data type is a synonym for CHAR.

CHARACTER VARYING(m,r) data type

The CHARACTER VARYING data type stores a string of letters, digits, and symbols of varying length, where *m* is the maximum size of the column (in bytes) and *r* is the minimum number of bytes reserved for that column.

The CHARACTER VARYING data type complies with ANSI/ISO standard for SQL; the non-ANSI VARCHAR data type supports the same functionality. For more information, see the description of the VARCHAR type in VARCHAR(m,r) data type on page 1417.

CLOB data type

The CLOB data type stores any kind of text data in random-access chunks, called sbspaces. Text data can include text-formatting information, if this information is also textual, such as PostScript™, Hypertext Markup Language (HTML), Standard Graphic Markup Language (SGML), or Extensible Markup Language (XML) data.

The term *smart large object* refers to CLOB and BLOB data types. The CLOB data type supports special operations for character strings that are inappropriate for BLOB values. A CLOB value can be up to 4 terabytes ($4*2^{40}$ bytes) in length. The minimum amount of disk space allocated for smart large object data types is 512 bytes.

Use the BLOB data type (see BLOB data type on page 1395) for random access to binary data. For general information about the CLOB and BLOB data types, see Smart large objects on page 1423.

The following SQL functions can perform operations on a CLOB column:

- FILETOCLOB copies a file into a CLOB column.
- LOTOFILE copies a CLOB (or BLOB) value into a file.
- LOCOPY copies a CLOB (or BLOB) value to a new smart large object.
- ENCRYPT_DES or ENCRYPT_TDES creates an encrypted BLOB value from a plain-text CLOB argument.
- DECRYPT_BINAR or DECRYPT_CHAR returns an unencrypted BLOB value from an encrypted BLOB argument (that ENCRYPT_DES or ENCRYPT_TDES created from a plain-text CLOB value).

For more information about these SQL functions, see the *HCL® Informix® Guide to SQL: Syntax*.

No casts exist for CLOB data. Therefore, the database server cannot convert data of the CLOB type to any other data type, except by using these encryption and decryption functions to return a BLOB. Within SQL, you are limited to the equality (=) comparison operation for CLOB data. To perform additional operations, you must use one of the application programming interfaces from within your client application.

Multibyte characters with CLOB

About this task

You can insert data into CLOB columns in the following ways:

- With the dbload or onload utilities
- With the LOAD statement (DB-Access)
- From CLOB (**ifx_lo_t**) host variables (ESQL/C)

For examples of CLOB types, see the *HCL® Informix® Guide to SQL: Tutorial* and the *IBM® Informix® Database Design and Implementation Guide*.

With GLS, the following rules apply:

- Multibyte CLOB characters must be defined in the database locale.
- The CLOB data type is collated in code-set order.
- The database server handles code-set conversions for CLOB data.

For more information about database locales, collation order, and code-set conversion, see the *HCL® Informix® GLS User's Guide*.

DATE data type

The DATE data type stores the calendar date. DATE data types require four bytes. A calendar date is stored internally as an integer value equal to the number of days since December 31, 1899.

Because DATE values are stored as integers, you can use them in arithmetic expressions. For example, you can subtract a DATE value from another DATE value. The result, a positive or negative INTEGER value, indicates the number of days that elapsed between the two dates. (You can use a UNITS DAY expression to convert the result to an INTERVAL DAY TO DAY data type.)

The following example shows the default display format of a DATE column:

```
mm/dd/yyyy
```

In this example, *mm* is the month (1-12), *dd* is the day of the month (1-31), and *yyyy* is the year (0001-9999). You can specify a different order of time units and a different time-unit separator than / (or no separator) by setting the **DBDATE** environment variable. For more information, see DBDATE environment variable on page 1469.

In non-default locales, you can display dates in culture-specific formats. The locale and the **GL_DATE** and **DBDATE** environment variables (as described in the next chapter) affect the display formatting of DATE values. They do not, however, affect the internal storage format for DATE columns in the database. For more information, see the *HCL® Informix® GLS User's Guide*.

DEC data type

The DEC data type is a synonym for DECIMAL.

DECIMAL

The DECIMAL data type can take two forms: DECIMAL(*p*) floating point and DECIMAL(*p*,*s*) fixed point.

In an ANSI-compliant database all DECIMAL numbers are fixed point.

By default, literal numbers that include a decimal (**.**) point are interpreted by the database server as DECIMAL values.

DECIMAL(p) Floating Point

The DECIMAL data type stores decimal floating-point numbers up to a maximum of 32 significant digits, where *p* is the total number of significant digits (the *precision*).

Specifying precision is optional. If you specify no precision (*p*), DECIMAL is treated as DECIMAL(16), a floating-point decimal with a precision of 16 places. DECIMAL(*p*) has an absolute exponent range between 10^{-130} and 10^{124}.

If you declare a DECIMAL(*p*) column in an ANSI-compliant database, the scale defaults to DECIMAL(*p*, 0), meaning that only integer values can be stored in this data type.

In a database that is not ANSI-compliant, a DECIMAL(*p*) is a floating-point data type of a scale large enough to store the exponential notation for a value.

For example, the following calculation shows how many bytes of storage a DECIMAL(5) column requires in the default locale (where the decimal point occupies a single byte):

1 byte for the sign of the data value
1 byte for the 1st digit
1 byte for the decimal point
4 bytes for the rest of the digits (precision of 5 - 1)
1 byte for the **e** symbol
1 byte for the sign of the exponent
3 bytes for the exponent
---
12 bytes total

Thus, "12345" in a DECIMAL(5) column is displayed as "12345.00000" (that is, with a scale of 6) in a database that is not ANSI-compliant.

DECIMAL (p,s) Fixed Point

In fixed-point numbers, DECIMAL(*p*,*s*), the decimal point is fixed at a specific place, regardless of the value of the number. When you specify a column of this type, you declare its precision (*p*) as the total number of digits that it can store, from 1 to 32. You declare its *scale* (*s*) as the total number of digits in the fractional part (that is, to the right of the decimal point).

All numbers with an absolute value less than $0.5 * 10^{-s}$ have the value zero. The largest absolute value of a DECIMAL(*p*,*s*) data type that you can store without an overflow error is $10^{p-s} - 10^{-s}$. A DECIMAL column typically stores numbers with fractional parts that must be stored and displayed exactly (for example, rates or percentages). In an ANSI-compliant database, all DECIMAL numbers must have absolute values in the range 10^{-32} to 10^{+31}.

DECIMAL Storage

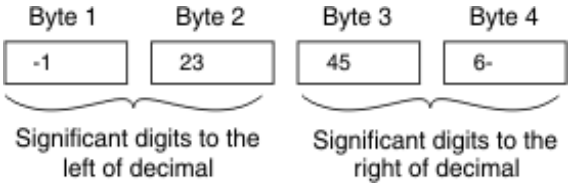
The database server uses one byte of disk storage to store two digits of a decimal number, plus an additional byte to store the exponent and sign, with the first byte representing a sign bit and a 7-bit exponent in excess-65 format. The rest of the bytes express the mantissa as base-100 digits. The significant digits to the left of the decimal and the significant digits to the right of the decimal are stored in separate groups of bytes. At the maximum *precision* specification, DECIMAL(32,s) data types can store s-1 decimal digits to the right of the decimal point, if s is an odd number.

How the database server stores decimal numbers is illustrated in the following example. If you specify DECIMAL(6,3), the data type consists of three significant digits in the integral part and three significant digits in the fractional part (for instance,

123.456). The three digits to the left of the decimal are stored on 2 bytes (where one of the bytes only holds a single digit) and the three digits to the right of the decimal are stored on another 2 bytes, as Figure 16: Schematic that illustrates the storage of digits in a decimal (p,s) value on page 1401 illustrates.

(The exponent byte is not shown.) With the additional byte required for the exponent and sign, DECIMAL(6,3) requires a total of 5 bytes of storage.

Figure 16. Schematic that illustrates the storage of digits in a decimal (p,s) value



You can use the following formulas (rounded down to a whole number of bytes) to calculate the byte storage (N) for a DECIMAL(*p*,*s*) data type (where N includes the byte that is required to store the exponent and the sign):

```
If the scale is odd: N = (precision + 4) / 2
If the scale is even: N = (precision + 3) / 2
```

For example, the data type DECIMAL(5,3) requires 4 bytes of storage (9/2 rounded down equals 4).

There is one caveat to these formulas. The maximum number of bytes the database server uses to store a decimal value is 17. One byte is used to store the exponent and sign, leaving 16 bytes to store up to 32 digits of precision. If you specify a precision of 32 and an *odd* scale, however, you lose 1 digit of precision. Consider, for example, the data type DECIMAL(32,31). This decimal is defined as 1 digit to the left of the decimal and 31 digits to the right. The 1 digit to the left of the decimal requires 1 byte of storage. This leaves only 15 bytes of storage for the digits to the right of the decimal. The 15 bytes can accommodate only 30 digits, so 1 digit of precision is lost.

DISTINCT data types

A DISTINCT type is a data type that is derived from a source type (called the base type).

A source type can be:

- A built-in type
- An existing DISTINCT type
- An existing named ROW type
- An existing opaque type

A DISTINCT type inherits from its source type the length and alignment on the disk. A DISTINCT type thus makes efficient use of the preexisting functionality of the database server.

When you create a DISTINCT data type, the database server automatically creates two explicit casts: one cast from the DISTINCT type to its source type and one cast from the source type to the DISTINCT type. A DISTINCT type based on a built-in source type does not inherit the built-in casts that are provided for the built-in type. A DISTINCT type does inherit, however, any user-defined casts that have been defined on the source type.

A DISTINCT type cannot be compared directly to its source type. To compare the two types, you must first explicitly cast one type to the other.

You must define a DISTINCT type in the database. Definitions of DISTINCT types are stored in the **sysxtdtypes** system catalog table. The following SQL statements maintain the definitions of DISTINCT types in the database:

- The CREATE DISTINCT TYPE statement adds a DISTINCT type to the database.
- The DROP TYPE statement removes a previously defined DISTINCT type from the database.

For more information about the SQL statements mentioned above, see the *HCL® Informix® Guide to SQL: Syntax*. For information about casting DISTINCT data types, see Casts for distinct types on page 1438. For examples that show how to create and register cast functions for a DISTINCT type, see the *IBM® Informix® Database Design and Implementation Guide*.

Size specifications in declarations of DISTINCT types whose base types are built-in character types can be affected by the SQL_LOGICAL_CHAR feature that is described in the section Logical Character Semantics in Character Type Declarations on page 1420.

DOUBLE PRECISION data types

The DOUBLE PRECISION keywords are a synonym for the FLOAT keyword.

Related information

FLOAT(n) on page 1402

FLOAT(n)

The FLOAT data type stores double-precision floating-point numbers with up to 17 significant digits. FLOAT corresponds to IEEE 4-byte floating-point, and to the **double** data type in C. The range of values for the FLOAT data type is the same as the range of the C **double** data type on your computer.

You can use *n* to specify the precision of a FLOAT data type, but SQL ignores the precision. The value *n* must be a whole number between 1 and 14.

A column with the FLOAT data type typically stores scientific numbers that can be calculated only approximately. Because floating-point numbers retain only their most significant digits, the number that you enter in this type of column and the number the database server displays can differ slightly.

The difference between the two values depends on how your computer stores floating-point numbers internally. For example, you might enter a value of 1.1000001 into a FLOAT field and, after processing the SQL statement, the database server might display this value as 1.1. This situation occurs when a value has more digits than the floating-point number can store. In this case, the value is stored in its approximate form with the least significant digits treated as zeros.

FLOAT data types usually require 8 bytes of storage per value. Conversion of a FLOAT value to a DECIMAL value results in 17 digits of precision.

Related information

DOUBLE PRECISION data types on page 1402

IDSSECURITYLABEL data type

The IDSSECURITYLABEL type stores a security label in a table that is protected by a label-based access control (LBAC) security policy.

Only a user who holds the **DBSECADM** role can create, alter, or drop a column of this data type. IDSSECURITYLABEL is a built-in DISTINCT OF VARCHAR(128) data type, but because its use is restricted to databases that implement label-based access control, it is not classified as a character data type. A table that is protected by a security policy can have only one IDSSECURITYLABEL column. A table that is not associated with any label-based security policy cannot include an IDSSECURITYLABEL column. You cannot encrypt the security label in a column of type IDSSECURITYLABEL.

For a discussion of security policies, security components, security labels, and other concepts of label-based access control (LBAC), see the HCL Informix® Security Guide.

INT data type

The INT data type is a synonym for INTEGER.

INT8

The INT8 data type stores whole numbers that can range in value from –9,223,372,036,854,775,807 to 9,223,372,036,854,775,807 [or -(2^{63}-1) to 2^{63}-1], for 18 or 19 digits of precision.

The number –9,223,372,036,854,775,808 is a reserved value that cannot be used. The INT8 data type is typically used to store large counts, quantities, and so on.

HCL Informix® stores INT8 data in internal format that can require up to 10 bytes of storage.

Arithmetic operations and sort comparisons are performed more efficiently on integer data than on floating-point or fixed-point decimal data, but INT8 cannot store data with absolute values beyond | 2^{63}-1 |. If a value exceeds the numeric range of INT8, the database server does not store the value.

INTEGER data type

The INTEGER data type stores whole numbers that range from -2,147,483,647 to 2,147,483,647 for 9 or 10 digits of precision.

The number 2,147,483,648 is a reserved value and cannot be used. The INTEGER value is stored as a signed binary integer and is typically used to store counts, quantities, and so on.

Arithmetic operations and sort comparisons are performed more efficiently on integer data than on float or decimal data. INTEGER columns, however, cannot store absolute values beyond (2^{31}-1). If a data value lies outside the numeric range of INTEGER, the database server does not store the value.

INTEGER data types require 4 bytes of storage per value.

LIST(e) data type

The LIST data type is a collection type that can store ordered non-NULL elements of the same SQL data type.

The LIST data type supports, but does not require, duplicate element values. The elements of a LIST data type have ordinal positions. The LIST object must have a first element, which can be followed by a second element, and so on.

For unordered collection data types that do not support ordinal positions, see MULTISET(e) data type on page 1406 and SET(e) data type on page 1414. For complex data types that can store a set of values that includes different SQL data types, see ROW Data Types on page 1432.

No more than 97 columns of the same table can be declared as LIST data types. (The same restriction applies to SET and MULTISET collection types.)

By default, the database server inserts new elements into a LIST object at the end of the set of elements. To support the ordinal position of a LIST, the INSERT statement provides the AT clause. This clause allows you to specify the position at which you want to insert a LIST element value. For more information, see the INSERT statement in the *HCL® Informix® Guide to SQL: Syntax*.

All elements in a LIST object have the same element type. To specify the element type, use the following syntax:

```
LIST(element_type NOT NULL)
```

The *element_type* of a LIST can be any of the following data types:

- A built-in type, except SERIAL, SERIAL8, BIGSERIAL, BYTE, and TEXT
- A DISTINCT type
- An unnamed or named ROW type
- Another collection type
- An opaque type

You must specify the NOT NULL constraint for LIST elements. No other constraints are valid for LIST columns. For more information about the syntax of the LIST data type, see the *HCL® Informix® Guide to SQL: Syntax*.

You can use LIST in most contexts where any other data type is valid. For example:

- After the IN predicate in the WHERE clause of a SELECT statement to search for matching LIST values
- As an argument to the CARDINALITY or **mi_collection_card( )** function to determine the number of elements in a LIST column

You *cannot* use LIST values as arguments to an aggregate function such as AVG, MAX, MIN, or SUM.

Just as with the other collection data types, you must use parentheses (`( )`) in data type declarations to delimit the set of elements of a LIST data type:

```
CREATE FUNCTION  update_nums( list1 LIST (ROW (a VARCHAR(10),
                                               b VARCHAR(10),
                                               c INT) NOT NULL ));
```

In SQL expressions that include literal LIST values, however, you must use braces (`{ }`) to delimit the set of elements of a LIST object, as in the examples that follow.

Two LIST values are equal if they have the same elements in the same order. The following are both examples of LIST objects, but their values are not equal. :

```
LIST{"blue", "green", "yellow"}
```

```
LIST{"yellow", "blue", "green"}
```

The above expressions are not equal because the values are not in the same order. To be equal, the second statement must be:

```
LIST{"blue", "green", "yellow"}
```

LVARCHAR(m) data type

Use the LVARCHAR data type to create a column for storing variable-length character strings whose upper limit (*m*) can be up to 32,739 bytes.

This limit is much greater than the VARCHAR data type, which is used for character strings that are no longer than 255 bytes.

The LVARCHAR data type is implemented as a built-in opaque data type. You can access LVARCHAR columns in remote tables by using distributed queries across databases of the same or different HCL Informix® instances.

By default, the database server interprets quoted strings as LVARCHAR types. It also uses LVARCHAR for input and output casts for opaque data types.

The LVARCHAR data type stores opaque data types in the string (external) format. Each opaque type has an input support function and cast, which convert it from LVARCHAR to a form that database servers can manipulate. Each opaque type also has an output support function and cast, which convert it from its internal representation to LVARCHAR.

Important: When LVARCHAR is declared (with no size specification) as the data type of a column in a database table, the default maximum size is 2 KB (2048 bytes), but you can specify an explicit maximum length of up to 32,739 bytes. When LVARCHAR is used in I/O operations on an opaque data type, however, the maximum size is limited only by the operating system.

You cannot use the MEDIUM or HIGH options of the UPDATE STATISTICS statement to calculate distribution statistics on LVARCHAR columns.

Size specifications in LVARCHAR data type declarations can be affected by the SQL_LOGICAL_CHAR feature that is described in the section Logical Character Semantics in Character Type Declarations on page 1420.

For more information about LVARCHAR, see the *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

MONEY(p,s) data type

The MONEY data type stores currency amounts.

TLike the DECIMAL(*p*,*s*) data type, MONEY can store fixed-point numbers up to a maximum of 32 significant digits, where *p* is the total number of significant digits (the precision) and *s* is the number of digits to the right of the decimal point (the scale).

Unlike the DECIMAL data type, the MONEY data type is always treated as a fixed-point decimal number. The database server defines the data type MONEY(*p*) as DECIMAL(*p*,2). If the precision and scale are not specified, the database server defines a MONEY column as DECIMAL(16,2).

You can use the following formula (rounded down to a whole number of bytes) to calculate the byte storage for a MONEY data type:

```
If the scale is odd: N = (precision + 4) / 2
If the scale is even: N = (precision + 3) / 2
```

For example, a MONEY data type with a precision of 16 and a scale of 2 (MONEY(16,2)) requires 10 or (16 + 3)/2, bytes of storage.

In the default locale, client applications format values from MONEY columns with the following currency notation:

- A currency symbol: a dollar sign (**$**) at the front of the value
- A thousands separator: a comma (**,**) that separates every three digits in the integer part of the value
- A decimal point: a period (**.**) between the integer and fractional parts of the value

To change the format for MONEY values, change the **DBMONEY** environment variable. For valid **DBMONEY** settings, see #unique_1962.

The default value that the database server uses for scale is locale-dependent. The default locale specifies a default scale of two. For non-default locales, if the scale is omitted from the declaration, the database server creates MONEY values with a locale-specific scale.

The currency notation that client applications use is locale-dependent. If you specify a nondefault locale, the client uses a culture-specific format for MONEY values that might differ from the default U.S. English format in the leading (or trailing) currency symbol, thousands separator, and decimal separator, depending on what the locale files specify. For more information about locale dependency, see the *HCL® Informix® GLS User's Guide*.

MULTISET(e) data type

The MULTISET data type is a collection type that stores a non-ordered set that can include duplicate element values.

The elements in a MULTISET have no ordinal position. That is, there is no concept of a first, second, or third element in a MULTISET. (For a collection type with ordinal positions for elements, see LIST(e) data type on page 1404.)

All elements in a MULTISET have the same element type. To specify the element type, use the following syntax:

```
MULTISET(element_type NOT NULL)
```

The *element_type* of a collection can be any of the following types:

- Any built-in type, except SERIAL, SERIAL8, BIGSERIAL, BYTE, and TEXT
- An unnamed or a named ROW type
- Another collection type or opaque type

You can use MULTISET anywhere that you use any other data type, unless otherwise indicated. For example:

- After the IN predicate in the WHERE clause of a SELECT statement to search for matching MULTISET values
- As an argument to the CARDINALITY or **mi_collection_card( )** function to determine the number of elements in a MULTISET column

You *cannot* use MULTISET values as arguments to an aggregate function such as AVG, MAX, MIN, or SUM.

You must specify the NOT NULL constraint for MULTISET elements. No other constraints are valid for MULTISET columns. For more information about the MULTISET collection type, see the *HCL® Informix® Guide to SQL: Syntax*.

Two multiset data values are equal if they have the same elements, even if the elements are in different positions within the set. The following examples are both multiset values but are not equal:

```
MULTISET {"blue", "green", "yellow"}
MULTISET {"blue", "green", "yellow", "blue"}
```

The following multiset values are equal:

```
MULTISET {"blue", "green", "blue", "yellow"}
MULTISET {"blue", "green", "yellow", "blue"}
```

No more than 97 columns of the same table can be declared as MULTISET data types. (The same restriction applies to SET and LIST collection types.)

Named ROW

See ROW data type, Named on page 1409.

NCHAR(n) data type

The NCHAR data type stores fixed-length character data. The data can be a string of single-byte or multibyte letters, digits, and other symbols that are supported by the code set of the database locale.

The main difference between CHAR and NCHAR data types is the collating order.

The collation order of the CHAR data type follows the code-set order, but the collating order of the NCHAR data type can be a localized order, if **DB_LOCALE** (or SET COLLATION) specifies a locale that defines a localized order for collation.

Size specifications ib NCHAR data type declarations can be affected by the SQL_LOGICAL_CHAR configuration parameter that is described in the section Logical Character Semantics in Character Type Declarations on page 1420.

In databases that are created with the NLSCASE INSENSITIVE property, operations on NCHAR strings ignore letter case, ordering data values without respect to or preference for letter case. For example, the NCHAR string `"IDS"` might precede or follow `"IdS"` or `"iDs"` in the collated list that a query returns, depending on the order in which these data strings are retrieved, because all of the following NCHAR strings are treated as duplicate values:

```
"ids" "IDS" "idS" "IDs" "IdS" "iDs" "iDS" "Ids"
```

NUMERIC(p,s) data type

The NUMERIC data type is a synonym for fixed-point DECIMAL.

NVARCHAR(m,r) data type

The NVARCHAR data type stores strings of varying lengths. The string can include digits, symbols, and both single-byte and (in some locales) multibyte characters.

The main difference between VARCHAR and NVARCHAR data types is the collation order. Collation of VARCHAR data follows code-set order, but NVARCHAR collation can be locale specific, if **DB_LOCALE** (or SET COLLATION) has specified a locale that defines a localized order for collation. (The section Collating VARCHAR Values on page 1419 describes an exception.)

A column declared as NVARCHAR, without parentheses or parameters, has a maximum size of one byte, and a reserved size of zero.

The first parameter in NVARCHAR data type declarations can be affected by the SQL_LOGICAL_CHAR configuration parameter that is described in the section Logical Character Semantics in Character Type Declarations on page 1420.

No more than 195 columns of the same table can be NVARCHAR data types.

In databases that are created with the NLSCASE INSENSITIVE property, operations on NVARCHAR strings ignore letter case, ordering data values without respect to or preference for letter case. For example, the NVARCHAR string `"IBM"` might precede or follow `"IbM"` or `"iBm"` in the collated list that a query returns, depending on the order in which these data strings are retrieved, because all of the following NVARCHAR strings are treated as duplicate values:

```
"ibm" "IBM" "ibM" "IBm" "IbM" "iBm" "iBM" "Ibm"
```

OPAQUE data types

An OPAQUE type is a data type for which you must provide information to the database server.

You must provide this information:

- A data structure for how the data values are stored on disk
- Support functions to determine how to convert between the disk storage format and the user format for data entry and display
- Secondary access methods that determine how the index on this data type is built, used, and manipulated
- User functions that use the data type
- A system catalog entry to register the OPAQUE type in the database

The internal structure of an OPAQUE type is not visible to the database server and can only be accessed through user-defined routines. Definitions for OPAQUE types are stored in the **sysxtdtypes** system catalog table. These SQL statements maintain the definitions of OPAQUE types in the database:

- The CREATE OPAQUE TYPE statement registers a new OPAQUE type in the database.
- The DROP TYPE statement removes a previously defined OPAQUE type from the database.

For more information about the above-mentioned SQL statements, see the *HCL® Informix® Guide to SQL: Syntax*. For information about how to create OPAQUE types and an example of an OPAQUE type, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

REAL data type

The REAL data type is a synonym for SMALLFLOAT.

ROW data type, Named

A named ROW data type must be declared with a name. This SQL identifier must be unique among data type names within the same database.

(An unnamed ROW type is a ROW type that contains fields but has no user-defined name.) Only named ROW types support data type inheritance. For more information, see ROW Data Types on page 1432.

Defining named ROW types

You must declare and register in the database a new named ROW type by using the CREATE ROW TYPE statement of SQL. Definitions for named ROW types are stored in the **sysxtdtypes** system catalog table.

The fields of a ROW data type can be any built-in data type or UDT, but TEXT or BYTE fields of a ROW type are valid in typed tables only. If you want to assign a ROW type to a column in the CREATE TABLE or ALTER TABLE statements, its elements cannot be TEXT or BYTE data types.

In general, the data type of a field of a ROW type can be any of these types:

- A built-in type (except for the TEXT or BYTE data types)
- A collection type (LIST, MULTISET, or SET)
- A distinct type
- Another named or unnamed ROW type
- An opaque type

These SQL statements maintain the definitions of named ROW data types:

- The CREATE ROW TYPE statement adds a named ROW type to the database.
- The DROP ROW TYPE statement removes a previously defined named ROW type from the database.

No more than 195 columns of the same table can be named ROW types.

For details about these SQL syntax statements, see the *HCL® Informix® Guide to SQL: Syntax*. For examples of how to create and use named ROW types, see the *IBM® Informix® Database Design and Implementation Guide*.

Equivalence and named ROW types

No two named ROW types can be equal, even if they have identical structures, because they have different names. For example, the following named ROW types have the same structure (the same number of fields and the same order of data types of fields within the row) but they are not equal:

```
name_t (lname CHAR(15), initial CHAR(1), fname CHAR(15))
emp_t (lname CHAR(15), initial CHAR(1), fname CHAR(15))
```

A Boolean equality condition like `name_t = emp_t` always evaluates to FALSE if both of the operands are different named ROW types.

Named ROW types and inheritance

Named ROW types can be part of a type-inheritance hierarchy. One named ROW type can be the parent (or supertype) of another named ROW type. A subtype in a hierarchy inherits all the properties of its supertype. Type inheritance is explained in the CREATE ROW TYPE statement in the *HCL® Informix® Guide to SQL: Syntax* and in the *IBM® Informix® Database Design and Implementation Guide*.

Typed tables

Tables that are part of an inheritance hierarchy must be typed tables. Typed tables are tables that have been assigned a named ROW type. For the syntax you use to create typed tables, see the CREATE TABLE statement in the *HCL® Informix® Guide to SQL: Syntax*. Table inheritance and its relation to type inheritance is also explained in that section. For information about how to create and use typed tables, see the *IBM® Informix® Database Design and Implementation Guide*.

ROW data type, Unnamed

An unnamed ROW type contains fields but has no user-declared name. An unnamed ROW type is defined by its structure.

Two unnamed ROW types are equal if they have the same structure (meaning the ordered list of the data types of the fields). If two unnamed ROW types have the same number of fields, and if the order of the data type of each field in one ROW type matches the order of data types of the corresponding fields in the other ROW data type, then the two unnamed ROW data types are equal.

For example, the following unnamed ROW types are equal:

```
ROW (lname char(15), initial char(1) fname char(15))
ROW (dept char(15), rating char(1) name char(15))
```

The following ROW types have the same number of fields and the same data types, but are not equal, because their fields are not in the same order:

```
ROW (x integer, y varchar(20), z real)
ROW (x integer, z real, y varchar(20))
```

A field of an unnamed ROW type can be any of the following data types:

- A built-in type
- A collection type
- A distinct type
- Another ROW type
- An opaque type

Unnamed ROW types cannot be used in typed tables or in type inheritance hierarchies. For more information about unnamed ROW types, see the *HCL® Informix® Guide to SQL: Syntax* and the *IBM® Informix® Database Design and Implementation Guide*.

Creating unnamed ROW types

About this task

You can create an unnamed ROW type in several ways:

- You can declare an unnamed ROW type using the ROW keyword. Each field in a ROW can have a different field type. To specify the field type, use the following syntax:

  ```
  ROW(field_name field_type, ...)
  ```

  The *field_name* must conform to the rules for SQL identifiers. (See the Identifier section in the *HCL® Informix® Guide to SQL: Syntax*.)

- To generate an unnamed ROW type, use the ROW keyword as a constructor with a series of values. A corresponding unnamed ROW type is created, using the default data types of the specified values.

  For example, the following declaration:

  ```
  ROW(1, 'abc', 5.30)
  ```

  defines this unnamed ROW data type:

  ```
  ROW (x INTEGER, y VARCHAR, z DECIMAL)
  ```

- You can create an unnamed ROW type by an implicit or explicit cast from a named ROW type or from another unnamed ROW type.
- The rows of any table (except a table defined on a named ROW type) are unnamed ROW types.

No more than 195 columns of the same table can be unnamed ROW types.

Inserting Values into Unnamed ROW Type Columns

About this task

When you specify field values for an unnamed ROW type, list the field values after the constructor and between parentheses. For example, suppose you have an unnamed ROW-type column. The following INSERT statement adds one group of field values to this ROW column:

```
INSERT INTO table1 VALUES (ROW(4, 'abc'))
```

You can specify a ROW column in the IN predicate in the WHERE clause of a SELECT statement to search for matching ROW values. For more information, see the Condition section in the *HCL® Informix® Guide to SQL: Syntax*.

SERIAL(n) data type

The SERIAL data type stores a sequential integer, of the INT data type, that is automatically assigned by the database server when a new row is inserted.

The default serial starting number is 1, but you can assign an initial value, *n*, when you create or alter the table.

- You must specify a positive number for the starting number.
- If you specify zero (0) for the starting number, the value that is used is the maximum positive value that already exists in the SERIAL column + 1.

The maximum value for SERIAL is 2,147,483,647. If you assign a number greater than 2,147,483,647, you receive a syntax error. Use the SERIAL8 or BIGSERIAL data type, rather than SERIAL, if you need a larger range.

A table can have no more than one SERIAL column, but it can have a SERIAL column and either a SERIAL8 column or a BIGSERIAL column.

SERIAL values in a column are not automatically unique. You must apply a unique index or primary key constraint to this column to prevent duplicate serial numbers. If you use the interactive schema editor in to define the table, a unique index is applied automatically to a SERIAL column.

SERIAL numbers might not be consecutive, because of concurrent users, rollbacks, and other factors.

The DEFINE *variable* LIKE *column* syntax of SPL for indirect typing declares a variable of the INTEGER data type if *column* is a SERIAL data type.

After a number is assigned, it cannot be changed. You can insert a value into a SERIAL column (using the INSERT statement) or reset a serial column (using the ALTER TABLE statement), if the new value does not duplicate any existing value in the column. To insert into a SERIAL column, your database server increments by one the previous value (or the reset value, if that is larger) and assigns the result as the entered value. If ALTER TABLE has reset the next value of a SERIAL column to a value smaller than values already in that column, however, the next value follows this formula:

```
(maximum existing value in SERIAL column) + 1
```

For example, if you reset the serial value of **customer.customer_num** to `50,` when the largest existing value is `128`, the next assigned number will be `129`. For more details on SERIAL data entry, see the *HCL® Informix® Guide to SQL: Syntax*.

A SERIAL column can store unique codes such as order, invoice, or customer numbers. SERIAL data values require four bytes of storage, and have the same precision as the INTEGER data type. For details of another way to assign unique whole numbers to each row of a database table, see the CREATE SEQUENCE statement in *HCL® Informix® Guide to SQL: Syntax*.

SERIAL8(n) data type

The SERIAL8 data type stores a sequential integer, of the INT8 data type, that is assigned automatically by the database server when a new row is inserted.

The SERIAL8 data type behaves like the SERIAL data type, but with a larger range. For more information about how to insert values into SERIAL8 columns, see the *HCL® Informix® Guide to SQL: Syntax*.

A SERIAL8 data column is commonly used to store large, unique numeric codes such as order, invoice, or customer numbers. SERIAL8 data values have the same precision and storage requirements as INT8 values (page INT8 on page 1403).

The default serial starting number is 1, but you can assign an initial value, *n*, when you create or alter the table.

- You must specify a positive number for the starting number.
- If you specify zero (0) for the starting number, the value that is used is the maximum positive value that already exists in the SERIAL8 column + 1.

A table can have no more than one SERIAL column, but it can have a SERIAL column and either a SERIAL8 column or a BIGSERIAL column.

SERIAL8 values in a column are not automatically unique. You must apply a unique index or primary key constraint to this column to prevent duplicate serial numbers. If you use the interactive schema editor in to define the table, a unique index is applied automatically to a SERIAL8 column.

SERIAL8 numbers might not be consecutive, because of concurrent users, rollbacks, and other factors.

The DEFINE *variable* LIKE *column* syntax of SPL for indirect typing declares a variable of the INTEGER data type if *column* is a SERIAL8 data type.

For more information, see Assigning a Starting Value for SERIAL8 on page 1413. For information about using the SERIAL8 data type with the INT8 or BIGINT data type, see Using SERIAL8 and BIGSERIAL with INT8 or BIGINT on page 1395

Assigning a Starting Value for SERIAL8

The default serial starting number is 1, but you can assign an initial value, *n*, when you create or alter the table. To start the values at `1` in a SERIAL8 column of a table, give the value `0` for the SERIAL8 column when you insert rows into that table. The database server will assign the value `1` to the SERIAL8 column of the first row of the table. The largest SERIAL8 value that you can assign is 2^{63}-1 (9,223,372,036,854,775,807). If you assign a value greater than this, you receive a syntax error. When the database server generates a SERIAL8 value of this maximum number, it wraps around and starts generating values beginning at `1`.

After a nonzero SERIAL8 number is assigned, it cannot be changed. You can, however, insert a value into a SERIAL8 column (using the INSERT statement) or reset the SERIAL8 value *n* (using the ALTER TABLE statement), if that value does not duplicate any existing values in the column.

When you insert a number into a SERIAL8 column or reset the next value of a SERIAL8 column, your database server assigns the next number in sequence to the number entered. If you reset the next value of a SERIAL8 column to a value that is less than the values already in that column, however, the next value is computed using the following formula:

```
maximum existing value in SERIAL8 column + 1
```

For example, if you reset the SERIAL8 value of the **customer_num** column in the **customer** table to `50`, when the highest-assigned customer number is 128, the next customer number assigned is 129.

For information about using the SERIAL8 data type with the INT8 or BIGINT data type, see Using SERIAL8 and BIGSERIAL with INT8 or BIGINT on page 1395

SET(e) data type

The SET data type is an unordered collection type that stores unique elements

Duplicate element values are not valid as explained in *HCL® Informix® Guide to SQL: Syntax*. (For a collection type that supports duplicate values, see the description of MULTISET in MULTISET(e) data type on page 1406.)

No more than 97 columns of the same table can be declared as SET data types. (The same restriction also applies to MULTISET and LIST collection types.)

The elements in a SET have no ordinal position. That is, no construct of a first, second, or third element in a SET exists. (For a collection type with ordinal positions for elements, see LIST(e) data type on page 1404.) All elements in a SET have the same element type. To specify the element type, use this syntax:

```
SET(element_type NOT NULL)
```

The *element_type* of a collection can be any of the following types:

- A built-in type, except SERIAL, SERIAL8, BIGSERIAL, BYTE, and TEXT
- A named or unnamed ROW type
- Another collection type
- An opaque type

You must specify the NOT NULL constraint for SET elements. No other constraints are valid for SET columns. For more information about the syntax of the SET collection type, see the *HCL® Informix® Guide to SQL: Syntax*.

You can use SET anywhere that you use any other data type, unless otherwise indicated. For example:

- After the IN predicate in the WHERE clause of a SELECT statement to search for matching SET values
- As an argument to the CARDINALITY or **mi_collection_card( )** function to determine the number of elements in a SET column

SET values are not valid as arguments to an aggregate function such as AVG, MAX, MIN, or SUM. For more information, see the Condition and Expression sections in the *HCL® Informix® Guide to SQL: Syntax*.

The following examples declare two sets. The first statement declares a set of integers and the second declares a set of character elements.

```
SET(INTEGER NOT NULL)
SET(CHAR(20) NOT NULL)
```

The following examples construct the same sets from value lists:

```
SET{1, 5, 13}
SET{"Oakland", "Menlo Park", "Portland", "Lenexa"}
```

In the following example, a SET constructor function is part of a CREATE TABLE statement:

```
CREATE TABLE tab
(
   c CHAR(5),
   s SET(INTEGER NOT NULL)
);
```

The following set values are equal:

```
SET{"blue", "green", "yellow"}
SET{"yellow", "blue", "green"}
```

SMALLFLOAT

The SMALLFLOAT data type stores single-precision floating-point numbers with approximately nine significant digits.

SMALLFLOAT corresponds to the **float** data type in C. The range of values for a SMALLFLOAT data type is the same as the range of values for the C **float** data type on your computer.

A SMALLFLOAT data type column typically stores scientific numbers that can be calculated only approximately. Because floating-point numbers retain only their most significant digits, the number that you enter in this type of column and the number the database displays might differ slightly depending on how your computer stores floating-point numbers internally.

For example, you might enter a value of 1.1000001 in a SMALLFLOAT field and, after processing the SQL statement, the application might display this value as 1.1. This difference occurs when a value has more digits than the floating-point number can store. In this case, the value is stored in its approximate form with the least significant digits treated as zeros.

SMALLFLOAT data types usually require four bytes of storage. Conversion of a SMALLFLOAT value to a DECIMAL value results in nine digits of precision.

SMALLINT data type

The SMALLINT data type stores small whole numbers that range from –32,767 to 32,767. The maximum negative number, –32,768, is a reserved value and cannot be used.

The SMALLINT value is stored as a signed binary integer.

Integer columns typically store counts, quantities, and so on. Because the SMALLINT data type requires only two bytes per value, arithmetic operations are performed efficiently. SMALLINT, however, stores only a limited range of values, compared to other built-in numeric data types. If a number is outside the range of the minimum and maximum SMALLINT values, the database server does not store the data value, but instead issues an error message.

TEXT data type

The TEXT data type stores any kind of text data. It can contain both single-byte and multibyte characters that the locale supports. The term *simple large object* refers to an instance of a TEXT or BYTE data type.

A TEXT column has a theoretical limit of 2^{31} bytes (two gigabytes) and a practical limit that your available disk storage determines. No more than 195 columns of the same table can be declared as TEXT data types. The same restriction also applies to BYTE data types.

You can store, retrieve, update, or delete the value in a TEXT column.

You can use TEXT operands in Boolean expressions only when you are testing for NULL values with the IS NULL or IS NOT NULL operators.

You can use the following methods, which can load rows or update fields, to insert TEXT data:

- With the **dbload** or **onload** utilities
- With the LOAD statement ()
- From TEXT host variables (ESQL)

A built-in cast exists to convert TEXT objects to CLOB objects. For more information, see the *IBM® Informix® Database Design and Implementation Guide*.

Strings of the TEXT data type are collated in code-set order. For more information about collating orders, see the *HCL® Informix® GLS User's Guide*.

Selecting data in a TEXT column

When you select a TEXT column, you can receive all or part of it. To retrieve it all, use the regular syntax for selecting a column. You can also select any part of a TEXT column by using subscripts, as this example shows:

```
SELECT cat_descr [1,75] FROM catalog WHERE catalog_num = 10001
```

The SELECT statement reads the first 75 bytes of the **cat_descr** column associated with the **catalog_num** value `10001`.

Loading data into a TEXT column

You can use the LOAD statement to insert data into a table. For example, the `inp.txt` file contains the following information:

```
1|aaaaa|
2|bbbbb|
3|cccccc|
```

To load this data into the `blobtab` table use the following statement:

```
LOAD FROM inp.txt INSERT INTO blobtab;
```

Limitations

You cannot use TEXT operands in arithmetic or string expressions, nor can you assign literals to TEXT columns in the SET clause of the UPDATE statement.

You also cannot use TEXT values in any of the following ways:

- With aggregate functions
- With the IN clause
- With the MATCHES or LIKE clauses
- With the GROUP BY clause
- With the ORDER BY clause

You cannot use a quoted text string, number, or any other actual value to insert or update TEXT columns.

You cannot use the MEDIUM or HIGH options of the UPDATE STATISTICS statement to calculate distribution statistics on TEXT columns.

Important: An error results if you try to return a TEXT column from a subquery, even if no TEXT column is used in a comparison condition or with the IN predicate.

Nonprintable Characters in TEXT Values

TEXT columns typically store documents, program source files, and so on. In the default U.S. English locale, data objects of type TEXT can contain a combination of printable ASCII characters and the following control characters:

- Tab (CTRL-I)
- New line (CTRL-J)
- New page (CTRL-L)

Both printable and nonprintable characters can be inserted in text columns. HCL Informix® products do not do any checking of data values that are inserted in a column of the TEXT data type. (Applications might have difficulty, however, in displaying TEXT values that include non-printable characters.) For detailed information about entering and displaying nonprintable characters, see Nonprintable Characters with CHAR.

Unnamed ROW

See ROW data type, Unnamed on page 1410.

VARCHAR(m,r) data type

The VARCHAR data type stores character strings of varying length that contain single-byte and (if the locale supports them) multibyte characters, where *m* is the maximum size (in bytes) of the column and *r* is the minimum number of bytes reserved for that column.

A column declared as VARCHAR without parentheses or parameters has a maximum size of one byte, and a reserved size of zero.

The VARCHAR data type is the HCL Informix® implementation of a character varying data type. The ANSI standard data type for varying-length character strings is CHARACTER VARYING.

The size of the maximum size (*m*) parameter of a VARCHAR column can range from 1 to 255 bytes. If you are placing an index on a VARCHAR column, the maximum size is 254 bytes. You can store character strings that are shorter, but not longer, than the *m* value that you specify.

Specifying the minimum reserved space (*r*) parameter is optional. This value can range from `0` to `255` bytes but must be less than the maximum size (*m*) of the VARCHAR column. If you do not specify any minimum value, it defaults to 0. You should specify this parameter when you initially intend to insert rows with short or NULL character strings in the column but later expect the data to be updated with longer values.

For variable-length strings longer than 255 bytes, you can use the LVARCHAR data type, whose upper limit is 32,739 bytes, instead of VARCHAR.

In an index based on a VARCHAR column (or on a NVARCHAR column), each index key has a length that is based on the data values that are actually entered, rather than on the declared maximum size of the column. (See, however, IFX_PAD_VARCHAR environment variable on page 1491 for information about how you can configure the effective size of VARCHAR and NVARCHAR data strings that HCL Informix® sends or receives.)

When you store a string in a VARCHAR column, only the actual data characters are stored. The database server does not strip a VARCHAR string of any user-entered trailing blanks, nor pad a VARCHAR value to the declared length of the column. If you specify a reserved space (*r*), but some data strings are shorter than *r* bytes, some space reserved for rows goes unused.

VARCHAR values are compared to other VARCHAR values (and to other character-string data types) in the same way that CHAR values are compared. The shorter value is padded on the right with blank spaces until the values have equal lengths; then they are compared for the full length.

No more than 195 columns of the same table can be VARCHAR data types.

Nonprintable Characters with VARCHAR

Nonprintable VARCHAR characters are entered, displayed, and treated in the same way that nonprintable characters in CHAR values are treated. For details, see Nonprintable Characters with CHAR.

Storing Numeric Values in a VARCHAR Column

When you insert a numeric value in a VARCHAR column, the stored value does not get padded with trailing blanks to the maximum length of the column. The number of digits in a numeric VARCHAR value is the number of characters that are required to store that value. For example, in the next example, the value stored in table **mytab** is `1`.

```
create table mytab (col1 varchar(10));
insert into mytab values (1);
```

Tip: VARCHAR treats C *null* (binary 0) and string terminators as termination characters for nonprintable characters.

In some East Asian locales, VARCHAR data types can store multibyte characters if the database locale supports a multibyte code set. If you store multibyte characters, make sure to calculate the number of bytes needed. For more information, see the *HCL® Informix® GLS User's Guide*.

Multibyte Characters with VARCHAR

The first parameter in VARCHAR data type declarations can be affected by the SQL_LOGICAL_CHAR feature that is described in the section Logical Character Semantics in Character Type Declarations on page 1420.

Collating VARCHAR Values

The main difference between the NVARCHAR and the VARCHAR data types (like the difference between CHAR and NCHAR) is the difference in collating order. In general, collation of VARCHAR (like CHAR and LVARCHAR) values is in the order of the characters as they exist in the code set.

An exception is the MATCHES operator, which applies a localized collation to NVARCHAR and VARCHAR values (and to CHAR, LVARCHAR, and NCHAR values) if you use bracket ([ ]) symbols to define ranges when **DB_LOCALE** (or SET COLLATION) has specified a localized collating order. For more information, see the *HCL® Informix® GLS User's Guide*.

Built-In Data Types

HCL Informix® supports the following built-in data types.

Category	Data Types
Character	CHAR, CHARACTER VARYING, LVARCHAR, NCHAR, NVARCHAR, VARCHAR, IDSSECURITYLABEL
Large-object	Simple-large-object types: BYTE, TEXT Smart-large-object types: BLOB, CLOB
Logical	BOOLEAN
Multirepresentational	BSON and JSON built-in opaque data types on page
Numeric	BIGINT, BIGSERIAL, DECIMAL, FLOAT, INT8, INTEGER, MONEY, SERIAL, SERIAL8, SMALLFLOAT, SMALLINT
Time	DATE, DATETIME, INTERVAL

Related information

Summary of data types on page 1389

BSON and JSON built-in opaque data types on page

Character Data Types

The character data types store string values.

Built-in Character Types

Table 74. Attributes of built-in character data types

	Size (in bytes)	Default	Reserved	Collation	Length
CHAR(n)	1 to 32,767	1 byte	None	Code set	Fixed
NCHAR(n)	1 to 32,767	1 byte	None	Localized	Fixed
VARCHAR(m, r)	1 to 255	0 for **r**	0 to 255 bytes	Code set	Variable
NVARCHAR(m, r)	1 to 255	0 for **r**	0 to 255 bytes	Localized	Variable
LVARCHAR(m)	1 to 32,739	2048 bytes	None	Code set	Variable

The database server also uses LVARCHAR to represent the external format of opaque data types. In I/O operations of the database server, LVARCHAR data values have no upper limit on their size, apart from file size restrictions or limits of your operating system or hardware resources.

Logical Character Semantics in Character Type Declarations

HCL Informix® supports a configuration parameter, SQL_LOGICAL_CHAR, whose setting can instruct the SQL parser to interpret the maximum size of character columns in data type declarations of the CREATE TABLE or ALTER TABLE statements as logical characters, rather than in units of bytes.

When a database is created, the current SQL_LOGICAL_CHAR setting for the database server is recorded in the **systables** table of the system catalog. The feature has no effect on tables that are subsequently created or altered in the database if the setting is OFF or 1.

In a database where the SQL_LOGICAL_CHAR setting is `ON` or is a digit between 2, 3, or 4, however, the SQL parser interprets explicit and implicit size declarations as logical characters in declarations of SPL variables and declarations of columns in database tables for the following character types:

- CHAR and CHARACTER
- CHARACTER VARYING and VARCHAR
- LVARCHAR
- NCHAR
- NVARCHAR
- DISTINCT types of the data types listed above
- DISTINCT types of those DISTINCT types
- ROW data type fields of the types listed above .
- LIST, MULTISET, and SET elements of the types listed above.

This feature has no effect on the maximum storage size limits for the character types listed in the previous table. For databases that use a multibyte locale, however, it can reduce the risk of data truncation when a string is inserted into a character column or assigned to a character variable.

For example, if 4 is the SQL_LOGICAL_CHAR setting for the database, then a VARCHAR(10, 5) specification is interpreted as requesting a maximum of 40 bytes of storage, with 5 of these bytes reserved, creating a VARCHAR(40, 5) data type in standard SQL notation, rather than what was specified in the declaration.

The reserve size parameters of VARCHAR and NVARCHAR data types are not affected by the SQL_LOGICAL_CHAR setting, because the minimum size of a multibyte character is 1 byte. In this example, the minimum size of 5 multibyte characters is 5 bytes, a size that remains unchanged.

See the description of the SQL_LOGICAL_CHAR configuration parameter in the *HCL® Informix® Administrator's Reference* for more information about the effect of the SQL_LOGICAL_CHAR setting in databases whose **DB_LOCALE** specifies a multibyte locale. For additional information about multibyte locales and logical characters, see the *HCL® Informix® GLS User's Guide*.

IDSSECURITYLABEL

HCL Informix® also supports the IDSSECURITYLABEL data type for systems that implement label-based access control (LBAC). This built-in data type can be formally classified as a character type, because it is defined as a DISTINCT OF VARCHAR(128) data type, but only users who hold the **DBSECADM** role can declare this data type in DDL operations. It supports the LBAC security feature, rather than functioning as a general-purpose character type.

Data Type Promotion

For some string-manipulation operations of HCL Informix®, the five built-in character data types listed above support data type promotion, in order to reduce the risk of string operations failing because a returned string is too large to be stored in an NVARCHAR or VARCHAR column or program variable. See the topic "Return Types from CONCAT and String Manipulation Functions" in *HCL® Informix® Guide to SQL: Syntax* for details of data type promotion among HCL Informix® character types.

National Language Support

The NCHAR and NVARCHAR types are sometimes called National Language Support data types because of their support for localized collation. Because columns of type VARCHAR or NVARCHAR have no default size, you must specify a size (no greater than 255) in their declaration. For VARCHAR or NVARCHAR columns on which an index is defined, the maximum size is 254 bytes.

NLSCASE INSENSITIVE Databases

In databases created with the `NLSCASE INSENSITIVE` keyword option, operations on data strings of the NCHAR or NVARCHAR types makes no distinction between uppercase and lowercase variants of the same letter. Rows are stored in NCHAR or NVARCHAR columns in the letter case in which characters were loaded, but data strings that consist of the same letters in the same sequence are evaluated as duplicates, even if the case of some letters is not identical. For example, the three NCHAR strings `"ABC"` and `"AbC"` and `"abC"` are treated as duplicates. Other built-in character types, including CHAR,

LVARCHAR, and VARCHAR, follow the default case-sensitive rules, so that the same three strings in a CHAR column evaluate to distinct values.

Databases with the `NLSCASE INSENSITIVE` property also ignore the letter case of DISTINCT data types whose base types are NCHAR or NVARCHAR, as well as NCHAR or NVARCHAR fields in named or unnamed ROW types, and NCHAR or NVARCHAR elements of COLLECTION data types, including LIST, SET, or MULTISET.

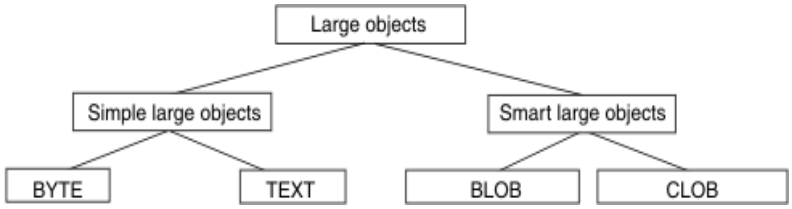
In a database that is insensitive to the letter case of NCHAR or NVARCHAR values, string manipulation operations might produce unexpected results, if they implicitly cast CHAR, LVARCHAR, or VARCHAR operands or arguments to NCHAR or NVARCHAR data types. In some contexts, the operation can return a duplicate string, despite letter case variations that the database server would not have treated as duplicates for the original data types.

Large-Object Data Types

A large object is a data object that is logically stored in a table column but physically stored independent of the column. Large objects are stored separate from the table because they typically store a large amount of data. Separation of this data from the table can increase performance.

Figure 17: Large-Object Data Types on page 1422 shows the large-object data types.

Figure 17. Large-Object Data Types



Only HCL Informix® supports BLOB and CLOB data types.

For the relative advantages and disadvantages of simple and smart large objects, see the *IBM® Informix® Database Design and Implementation Guide*.

Simple Large Objects

Simple large objects are a category of large objects that have a theoretical size limit of 2^{31} bytes and a practical limit that your disk capacity determines. HCL Informix® supports these simple-large-object data types:

BYTE

Stores binary data. For more detailed information about this data type, see the description on page BYTE data type on page 1396.

TEXT

Stores text data. For more detailed information about this data type, see the description on page TEXT data type on page 1415.

No more than 195 columns of the same table can be declared as BYTE or TEXT data types. Unlike smart large objects, simple large objects do not support random access to the data. When you transfer a simple large object between a client

application and the database server, you must transfer the entire BYTE or TEXT value. If the data cannot fit into memory, you must store the data value in an operating-system file and then retrieve it from that file.

The database server stores simple large objects in *blobspaces*. A *blobspace* is a logical storage area that contains one or more chunks that only store BYTE and TEXT data. For information about how to define blobspaces, see your *HCL® Informix® Administrator's Guide*.

Smart large objects

Smart large objects are a category of large objects that support random access to the data, and that are generally recoverable.

The random access feature allows you to seek and read through the smart large object as if it were an operating-system file.

Smart large objects are also useful for opaque data types with large storage requirements. (See the description of opaque data types in Opaque Data Types on page 1433.) They have a theoretical size limit of 2^{42} bytes and a practical limit that your disk capacity determines.

HCL Informix® supports the following smart-large-object data types:

BLOB

Stores binary data. For more information about this data type, see the description on page BLOB data type on page 1395.

CLOB

Stores text data. For more information about this data type, see CLOB data type on page 1398.

HCL Informix® stores smart large objects in *sbspaces*. An *sbspace* is a logical storage area that contains one or more chunks that store only BLOB and CLOB data. For information about how to define sbspaces, see your .

When you define a BLOB or CLOB column, you can determine the following large-object characteristics:

- LOG and NOLOG: whether the database server should log the smart large object in accordance with the current database logging mode.
- KEEP ACCESS TIME and NO KEEP ACCESS TIME: whether the database server should keep track of the last time the smart large object was accessed.
- HIGH INTEG and MODERATE INTEG: whether the database server should use sbspace page headers and page footers to detect data corruption (HIGH INTEG), or only use page headers (MODERATE INTEG).

Use of these characteristics can affect performance. For information, see your .

When an SQL statement accesses a smart-large-object, the database server does not send the actual BLOB or CLOB data. Instead, it establishes a pointer to the data and returns this pointer. The client application can then use this pointer in open, read, or write operations on the smart large object.

To access a BLOB or CLOB column from within a client application, use one of the following application programming interfaces (APIs):

- From within programs, use the smart-large-object API. (For more information, see the *HCL® Informix® Enterprise Replication Guide*.)
- From within a module, use the Client and Server API. (For more information, see the *HCL® Informix® DataBlade® API Programmer's Guide*.)

For information about smart large objects, see the *HCL® Informix® Guide to SQL: Syntax* and *IBM® Informix® Database Design and Implementation Guide*.

Time Data Types

DATE and DATETIME data values represent zero-dimensional points in time; INTERVAL data values represent 1-dimensional spans of time, with positive or negative values. DATE precision is always an integer count of days, but various field qualifiers can define the DATETIME and INTERVAL precision. You can use DATE, DATETIME, and INTERVAL data in arithmetic and relational expressions. You can manipulate a DATETIME value with another DATETIME value, an INTERVAL value, the current time (specified by the keyword CURRENT), or some unit of time (using the keyword UNITS).

You can use a DATE value in most contexts where a DATETIME value is valid, and vice versa. You also can use an INTERVAL operand in arithmetic operations where a DATETIME value is valid. In addition, you can add two INTERVAL values and multiply or divide an INTERVAL value by a number.

An INTERVAL column can hold a value that represents the difference between two DATETIME values or the difference between (or sum of) two INTERVAL values. In either case, the result is a span of time, which is an INTERVAL value. Conversely, if you add or subtract an INTERVAL from a DATETIME value, another DATETIME value is produced, because the result is a specific time.

Table 75: Arithmetic Operations on DATE, DATETIME, and INTERVAL Values on page 1424 lists the binary arithmetic operations that you can perform on DATE, DATETIME, and INTERVAL operands, and the data type that is returned by the arithmetic expression.

Table 75. Arithmetic Operations on DATE, DATETIME, and INTERVAL Values

Operand 1	Operator	Operand 2	Result
DATE	-	DATETIME	INTERVAL
DATETIME	-	DATE	INTERVAL
DATE	+ *or* -	INTERVAL	DATETIME
DATETIME	-	DATETIME	INTERVAL
DATETIME	+ *or* -	INTERVAL	DATETIME
INTERVAL	+	DATETIME	DATETIME
INTERVAL	+ *or* -	INTERVAL	INTERVAL
DATETIME	-	CURRENT	INTERVAL
CURRENT	-	DATETIME	INTERVAL

Table 75. Arithmetic Operations on DATE, DATETIME, and INTERVAL Values (continued)

Operand 1	Operator	Operand 2	Result
INTERVAL	**+**	CURRENT	DATETIME
CURRENT	**+** *or* **-**	INTERVAL	DATETIME
DATETIME	**+** *or* **-**	UNITS	DATETIME
INTERVAL	**+** *or* **-**	UNITS	INTERVAL
INTERVAL	* *or* /	NUMBER	INTERVAL

No other combinations are allowed. You cannot add two DATETIME values because this operation does not produce either a specific time or a span of time. For example, you cannot add December 25 and January 1, but you can subtract one from the other to find the time span between them.

Manipulating DATETIME Values

You can subtract most DATETIME values from each other.

Dates can be in any order and the result is either a positive or a negative INTERVAL value. The first DATETIME value determines the precision of the result, which includes the same time units as the first operand.

If the second DATETIME value has fewer fields than the first, the precision of the second operand is increased automatically to match the first.

In the following example, subtracting the DATETIME YEAR TO HOUR value from the DATETIME YEAR TO MINUTE value results in a positive interval value of 60 days, 1 hour, and 30 minutes. Because minutes were not included in the second operand, the database server sets the minutes value for the second operand to 0 before performing the subtraction.

```
DATETIME (2003-9-30 12:30) YEAR TO MINUTE
   - DATETIME (2003-8-1 11) YEAR TO HOUR

Result: INTERVAL (60 01:30) DAY TO MINUTE
```

If the second DATETIME operand has more fields than the first (regardless of whether the precision of the extra fields is larger or smaller than those in the first operand), the additional time unit fields in the second value are ignored in the calculation.

In the next expression (and its result), the year is not included for the second operand. Therefore, the year is set automatically to the current year (from the system clock-calendar), in this example 2005, and the resulting INTERVAL is negative, which indicates that the second date is later than the first.

```
DATETIME (2005-9-30) YEAR TO DAY
   - DATETIME (10-1) MONTH TO DAY

Result: INTERVAL (-1) DAY TO DAY [assuming that the current
                                  year is 2005]
```

You can compare two DATETIME values by using the mi_datetime_compare() function.

Related information

#unique_1975

The mi_datetime_compare() function on page

Manipulating DATETIME with INTERVAL Values

INTERVAL values can be added to or subtracted from DATETIME values. In either case, the result is a DATETIME value. If you are adding an INTERVAL value to a DATETIME value, the order of values is unimportant; however, if you are subtracting, the DATETIME value must come first. Adding or subtracting a positive INTERVAL value moves the DATETIME result forward or backward in time. The expression shown in the following example moves the date ahead by three years and five months:

```
DATETIME (2000-8-1) YEAR TO DAY
   + INTERVAL (3-5) YEAR TO MONTH

Result: DATETIME (2004-01-01) YEAR TO DAY
```

Important: Evaluate the logic of your addition or subtraction. Remember that months can have 28, 29, 30, or 31 days and that years can have 365 or 366 days.

In most situations, the database server automatically adjusts the calculation when the operands do not have the same precision. In certain contexts, however, you must explicitly adjust the precision of one value to perform the calculation. If the INTERVAL value you are adding or subtracting has fields that are not included in the DATETIME value, you must use the EXTEND function to increase the precision of the DATETIME value. (For more information about the EXTEND function, see the Expression segment in the *HCL® Informix® Guide to SQL: Syntax*.)

For example, you cannot subtract an INTERVAL MINUTE TO MINUTE value from the DATETIME value in the previous example that has a YEAR TO DAY field qualifier. You can, however, use the EXTEND function to perform this calculation, as the following example shows:

```
EXTEND (DATETIME (2008-8-1) YEAR TO DAY, YEAR TO MINUTE)
   - INTERVAL (720) MINUTE(3) TO MINUTE

Result: DATETIME (2008-07-31 12:00) YEAR TO MINUTE
```

The EXTEND function allows you to explicitly increase the DATETIME precision from YEAR TO DAY to YEAR TO MINUTE. This allows the database server to perform the calculation, with the resulting extended precision of YEAR TO MINUTE.

Manipulating DATE with DATETIME and INTERVAL Values

You can use DATE operands in some arithmetic expressions with DATETIME or INTERVAL operands by writing expressions to do the manipulating, as Table 76: Results of Expressions That Manipulate DATE with DATETIME or INTERVAL Values on page 1427 shows.

Table 76. Results of Expressions That Manipulate DATE with DATETIME or INTERVAL Values

Expression	Result
DATE **–** DATETIME	INTERVAL
DATETIME **–** DATE	INTERVAL
DATE **+** *or* **–** INTERVAL	DATETIME

In the cases that Table 76: Results of Expressions That Manipulate DATE with DATETIME or INTERVAL Values on page 1427 shows, DATE values are first converted to their corresponding DATETIME equivalents, and then the expression is evaluated by the rules of arithmetic.

Although you can interchange DATE and DATETIME values in many situations, you must indicate whether a value is a DATE or a DATETIME data type. A DATE value can come from the following sources:

- A column or program variable of type DATE
- The TODAY keyword
- The DATE( ) function
- The MDY function
- A DATE literal

A DATETIME value can come from the following sources:

- A column or program variable of type DATETIME
- The CURRENT keyword
- The EXTEND function
- A DATETIME literal

The database locale defines the default DATE and DATETIME formats. For the default locale, U.S. English, these formats are '*mm*/*dd*/*yy*' for DATE values and '*yyyy-mm-dd hh*:MM:*ss*' for DATETIME values.

To represent DATE and DATETIME values as character strings, the fields in the strings must be in the required order. In other words, when a DATE value is expected, the string must be in DATE format and when a DATETIME value is expected, the string must be in DATETIME format. For example, you can use the string `10/30/2008` as a DATE string but not as a DATETIME string. Instead, you must use `2008-10-30` or `08-10-30` as the DATETIME string.

In a nondefault locale, literal DATE and DATETIME strings must match the formats that the locale defines. For more information, see the *HCL® Informix® GLS User's Guide*.

You can customize the DATE format that the database server expects with the **DBDATE** and **GL_DATE** environment variables. You can customize the DATETIME format that the database server expects with the **DBTIME** and **GL_DATETIME** environment variables. For more information, see DBDATE environment variable on page 1469 and #unique_1978. For more information about all these environment variables, see the *HCL® Informix® GLS User's Guide*.

You can also subtract one DATE value from another DATE value, but the result is a positive or negative INTEGER count of days, rather than an INTERVAL value. If an INTERVAL value is required, you can either use the UNITS DAY operator to convert the INTEGER value into an INTERVAL DAY TO DAY value, or else use EXTEND to convert one of the DATE values into a DATETIME value before subtracting.

For example, the following expression uses the **DATE**( ) function to convert character string constants to DATE values, calculates their difference, and then uses the UNITS DAY keywords to convert the INTEGER result into an INTERVAL value:

```
(DATE ('5/2/2007') - DATE ('4/6/1968')) UNITS DAY

Result: INTERVAL (12810) DAY(5) TO DAY
```

Important: Because of the high precedence of UNITS relative to other SQL operators, you should generally enclose any arithmetic expression that is the operand of UNITS within parentheses, as in the preceding example.

If you need YEAR TO MONTH precision, you can use the EXTEND function on the first DATE operand, as the following example shows:

```
EXTEND (DATE ('5/2/2007'), YEAR TO MONTH) - DATE ('4/6/1969')

Result: INTERVAL (39-01) YEAR TO MONTH
```

The resulting INTERVAL precision is YEAR TO MONTH, because the DATETIME value came first. If the DATE value had come first, the resulting INTERVAL precision would have been DAY(5) TO DAY.

Related information

#unique_1975

#unique_1979

Manipulating INTERVAL Values

You can add or subtract INTERVAL values only if both values are from the same class; that is, if both are year-month or both are day-time.

In the following example, a SECOND TO FRACTION value is subtracted from a MINUTE TO FRACTION value:

```
INTERVAL (100:30.0005) MINUTE(3) TO FRACTION(4)
   - INTERVAL (120.01) SECOND(3) TO FRACTION

Result: INTERVAL (98:29.9905) MINUTE TO FRACTION(4)
```

The use of numeric qualifiers alerts the database server that the MINUTE and FRACTION in the first value and the SECOND in the second value exceed the default number of digits.

When you add or subtract INTERVAL values, the second value cannot have a field with greater precision than the first. The second INTERVAL, however, can have a field of smaller precision than the first. For example, the second INTERVAL can be

HOUR TO SECOND when the first is DAY TO HOUR. The additional fields (in this case MINUTE and SECOND) in the second INTERVAL value are ignored in the calculation.

You can compare two INTERVAL values by using the mi_interval_compare() function.

Related information

#unique_1979

The mi_interval_compare() function on page

Multiplying or Dividing INTERVAL Values

You can multiply or divide INTERVAL values by numbers. Any remainder from the calculation is ignored, however, and the result is truncated to the precision of the INTERVAL. The following expression multiplies an INTERVAL value by a literal number that has a fractional part:

```
INTERVAL (15:30.0002) MINUTE TO FRACTION(4) * 2.5

Result: INTERVAL (38:45.0005) MINUTE TO FRACTION(4)
```

In this example, 15 * 2.5 = 37.5 minutes, 30 * 2.5 = 75 seconds, and 2 * 2.5 = 5 FRACTION (4). The 0.5 minute is converted into 30 seconds and 60 seconds are converted into 1 minute, which produces the final result of 38 minutes, 45 seconds, and 0.0005 of a second. The result of any calculation has the same precision as the original INTERVAL operand.

Extended Data Types

HCL Informix® enables you to create *extended data types* to characterize data that cannot easily be represented with the built-in data types. (You cannot, however, use extended data types in distributed transactions that query external tables.) You can create these categories of extended data types:

- Complex data types
- Distinct data types
- Opaque data types

Sections that follow provide an overview of each of these data types.

For more information about extended data types, see the *IBM® Informix® Database Design and Implementation Guide* and *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

Related information

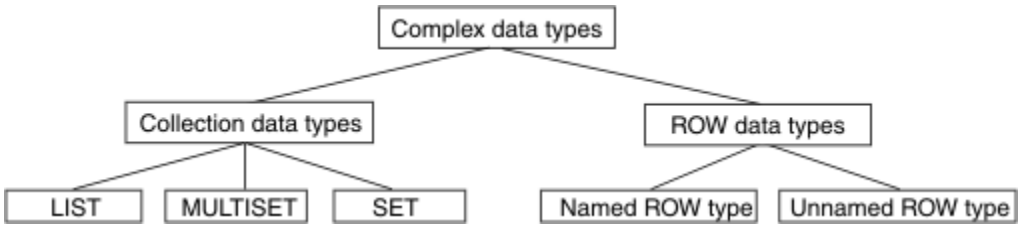
Summary of data types on page 1389

Complex data types

A *complex data type* can store one or more values of other built-in or extended data types.

Figure 18: Complex Data Types of HCL Informix on page 1430 shows the complex types that HCL Informix® supports.

Figure 18. Complex Data Types of HCL Informix®



The following table summarizes the structure of the complex data types.

Table 77. Collection types are complex data types that are made up of elements, each of which is of the same data type.

Collection types	Description
LIST	A group of ordered elements, each of which need not be unique within the group.
MULTISET	A group of elements, each of which need not be unique. The order of the elements is ignored.
SET	A group of elements, each of which is unique. The order of the elements is ignored.

Table 78. ROW types are complex data types that are made up of fields.

ROW types	Description
Named ROW type	Row types that are identified by their name.
Unnamed ROW type	Row types that are identified by their structure.

Complex data types can be nested. For example, you can construct a ROW type whose fields include one or more sets, multisets, ROW types, and lists. Likewise, a collection type can have elements whose data type is a ROW type or a collection type.

Complex types that include opaque types inherit the following support functions.

- **input**
- **output**
- **send**
- **recv**
- **import**
- **export**
- **import_binary**
- **export_binary**
- **assign**
- **destroy**
- **LO_handles**
- **hash**

- **lessthan**
- **equal**
- **lessthan** (for ROW types only)

The topics that follow summarize the complex data types. For more information, see the *IBM® Informix® Database Design and Implementation Guide*.

Collection Data Types

A collection data type is a complex type that is made up of one or more elements, all of the same data type. A collection element can be of any data type (including other complex types) except BYTE, TEXT, SERIAL, SERIAL8, or BIGSERIAL.

Important: An element cannot have a *NULL* value. You must specify the *NOT NULL* constraint for collection elements. No other constraints are valid for collections.

HCL Informix® supports three kinds of built-in collection types: LIST, SET, and MULTISET. The keywords used to declare these collections are the names of the *type constructors* or just *constructors*. For the syntax of collection types, see the *HCL® Informix® Guide to SQL: Syntax*. No more than 97 columns of the same table can be declared as collection data types.

When you specify element values for a collection, list the element values after the constructor and between braces ({ }). For example, suppose you have a collection column with the following MULTISET data type:

```
CREATE TABLE table1
(
   mset_col MULTISET(INTEGER NOT NULL)
)
```

The next INSERT statement adds one group of element values to this column. (The word MULTISET in these two examples is the MULTISET constructor.)

```
INSERT INTO table1 VALUES (MULTISET{5, 9, 7, 5})
```

You can leave the braces empty to indicate an empty set:

```
INSERT INTO table1 VALUE (MULTISET{})
```

An empty collection is not equivalent to a NULL value for the column.

Accessing collection data

About this task

To access the elements of a collection column, you must fetch the collection into a collection variable and modify the contents of the collection variable. Collection variables can be either of the following types:

- Variables in an SPL routine

  For more information, see the *HCL® Informix® Guide to SQL: Tutorial*.

- Host variables in programs

  For more information, see the *HCL® Informix® Enterprise Replication Guide*.

You can also use nested dot notation to access collection data. For more about accessing elements of a collection, see the *HCL® Informix® Guide to SQL: Tutorial*.

Important: Collection data types are not valid as arguments to functions that are used for functional indexes.

ROW Data Types

A ROW data type is an ordered collection of one or more elements, called *fields*. Each field has a name and a data type. The fields of a ROW are comparable to the columns of a table, but with important differences:

- A field has no default clause.
- You cannot define constraints on a field.
- You can only use fields with row types, not with tables.

Two kinds of ROW data types exist:

- *Named ROW data types* are identified by their names.
- *Unnamed ROW data types* are identified by their structure.

The *structure* of an unnamed ROW data type is the number (and the order of data types) of its fields.

No more than 195 columns of the same table can be declared as ROW data types. For more information about ROW data types, see ROW data type, Named on page 1409 and ROW data type, Unnamed on page 1410.

You can cast between named and unnamed ROW data types; this is described in the *IBM® Informix® Database Design and Implementation Guide*.

Distinct Data Types

A distinct data type has the same internal structure as some other source data type in the database. The source type can be a built-in or extended data type. What distinguishes a distinct type from its source type are support functions that are defined on the distinct type.

No more than approximately 97 columns of the same table can be DISTINCT of collection data types (SET, LIST, and MULTISET). No more than approximately 195 columns of the same table can be DISTINCT types that are based on BYTE, TEXT, ROW, LVARCHAR, NVARCHAR, or VARCHAR source types. (Here 195 columns is an approximate lower limit that applies to platforms with a 2 Kb base page size. For platforms with a base page size of 4 Kb, such as Windows™ and AIX® systems, the upper limit is approximately 450 columns of these data types.) For more information, see the section DISTINCT data types on page 1401. See also *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

Opaque Data Types

An opaque data type is a user-defined or built-in data type that is fully encapsulated. The internal structure of an opaque data type is unknown to the database server.

Except for user-defined types (UDTs) that are DISTINCT of built-in non-opaque types, UDTs whose source types are built-in types are opaque data types. Similarly, UDTs that are DISTINCT of built-in opaque types are opaque types.

Built-in opaque data types

The built-in data types BLOB, BOOLEAN, CLOB, BSON, JSON, and LVARCHAR are implemented as opaque data types. You can access all of these in other databases of the same Informix® instance, but you cannot access the BLOB or CLOB built-in opaque data types in cross-server distributed operations.

UDTs that are DISTINCT of built-in opaque types and that are cast to built-in types are valid in cross-server queries and other DML operations, but all the casts and all the DISTINCT OF definitions for the UDTs must be identical in every participating database.

Several system catalog tables, whose schema are shown in Structure of the System Catalog on page 1323, have columns of built-in opaque data types. For information on how the system catalog emcodes columns of built-in opaque data types, see SYSCOLUMNS on page 1336.

User-defined opaque data types

You must provide the following information to the database server for an opaque data type:

- A data structure for how the data values are stored on disk
- Support functions to determine how to convert between the disk storage format and the user format for data entry and display
- Secondary access methods that determine how the index on this data type is built, used, and manipulated
- User functions that use the data type
- A system catalog entry to register the opaque type in the database

The internal structure of an opaque type is not visible to the database server and can only be accessed through user-defined routines. Definitions for opaque types are stored in the sysxtdtypes system catalog table. These SQL statements maintain the definitions of opaque types in the database:

- The CREATE OPAQUE TYPE statement registers a new opaque type in the database.
- The DROP TYPE statement removes a previously defined opaque type from the database.

For more information, see the section OPAQUE data types on page 1408. See also *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

Data Type Casting and Conversion

About this task

Occasionally, the data type that was assigned to a column with the CREATE TABLE statement is inappropriate. You can change the data type of a column when you are required to store larger values than the current data type can accommodate. The database server allows you to change the data type of the column or to cast its values to a different data type with either of the following methods:

- Use the ALTER TABLE statement to modify the data type of a column.

  For example, if you create a SMALLINT column and later find that you must store integers larger than 32,767, you must change the data type of that column to store the larger value. You can use ALTER TABLE to change the data type to INTEGER. The conversion changes the data type of all values that currently exist in the column and any new values that might be added.
- Use the CAST AS keywords or the double colon (::) cast operator to cast a value to a different data type.

  Casting does not permanently alter the data type of a value; it expresses the value in a more convenient form. Casting user-defined data types into built-in types allows client programs to manipulate data types without knowledge of their internal structure.

If you change data types, the new data type must be able to store all of the old value.

Both data-type conversion and casting depend on casts registered in the **syscasts** system catalog table. For information about **syscasts**, see SYSCASTS on page 1332.

A cast is either built-in or user defined. Guidelines exist for casting distinct and extended data types. For more information about casting opaque data types, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*. For information about casting other extended data types see, the *IBM® Informix® Database Design and Implementation Guide*.

Using Built-in Casts

User **informix** owns built-in casts. They govern conversions from one built-in data type to another. Built-in casts allow the database server to attempt the following data-type conversions:

- A character type to any other character type
- A character type to or from another built-in type
- A numeric type to any other numeric type

The database server automatically invokes appropriate built-in casts when required. For time data types, conversion between DATE and DATETIME data types requires explicit casts with the EXTEND function, and explicit casts with the UNITS operator are required for number-to-INTERVAL conversion. Built-in casts are unavailable for converting large (BYTE, BLOB, CLOB, and TEXT) built-in types to other built-in data types.

When you convert a column from one built-in data type to another, the database server applies the appropriate built-in casts to each value already in the column. If the new data type cannot store any of the resulting values, the ALTER TABLE statement fails.

For example, if you try to convert a column from the INTEGER data type to the SMALLINT data type and the following values exist in the INTEGER column, the database server does not change the data type, because SMALLINT columns cannot accommodate numbers greater than 32,767:

```
100    400    700    50000      700
```

The same situation might occur if you attempt to transfer data from FLOAT or SMALLFLOAT columns to INTEGER, SMALLINT, or DECIMAL columns. Errors of overflow, underflow, or truncation can occur during data type conversion.

Sections that follow describe database server behavior during certain types of casts and conversions.

Converting from number to number

When you convert data from one number data type to another, you occasionally find rounding errors.

The following table indicates which numeric data type conversions are acceptable and what kinds of errors you can encounter when you convert between certain numeric data types. In the table, the following codes are used:

OK

No error

P

An error can occur, depending on the precision of the decimal

E

An error can occur, depending on the data value

D

No error, but less significant digits might be lost

Table 79. Acceptable conversions and possible errors

Target Type	SMALL INT	INTEGER	INT8	SMALL FLOAT	FLOAT	DECIMAL
SMALLINT	OK	OK	OK	OK	OK	OK
INTEGER	E	OK	OK	E	OK	P
INT8	E	E	OK	D	E	P
SMALLFLOAT	E	E	E	OK	OK	P
FLOAT	E	E	E	D	OK	P
DECIMAL	E	E	E	D	D	P

For example, if you convert a FLOAT value to DECIMAL(4,2), your database server rounds off the floating-point number before storing it as DECIMAL.

This conversion can result in an error depending on the precision assigned to the DECIMAL column.

Converting Between Number and Character

You can convert a character column (of a data type such as CHAR, NCHAR, NVARCHAR, or VARCHAR) to a numeric column. If a data string, however, contains any characters that are not valid in a number column (for example, the letter *l* instead of the number *1*), the database server returns an error.

You can also convert a numeric column to a character column. If the character column is not large enough to receive the number, however, the database server generates an error. If the database server generates an error, it cannot complete the ALTER TABLE statement or cast, and leaves the column values as characters. You receive an error message and the statement is rolled back automatically (regardless of whether you are in a transaction).

Converting Between INTEGER and DATE

You can convert an integer column (SMALLINT, INTEGER, or INT8) to a DATE value. The database server interprets the integer as a value in the internal format of the DATE column. You can also convert a DATE column to an integer column. The database server stores the internal format of the DATE column as an integer representing a Julian date.

Converting Between DATE and DATETIME

You can convert DATE columns to DATETIME columns. If the DATETIME column contains more fields than the DATE column, however, the database server either ignores the fields or fills them with zeros. The illustrations in the following list show how these two data types are converted (assuming that the default date format is *mm*/*dd*/*yyyy*):

- If you convert DATE to DATETIME YEAR TO DAY, the database server converts the existing DATE values to DATETIME values. For example, the value 08/15/2002 becomes 2002-08-15.
- If you convert DATETIME YEAR TO DAY to the DATE format, the value 2002-08-15 becomes 08/15/2002.
- If you convert DATE to DATETIME YEAR TO SECOND, the database server converts existing DATE values to DATETIME values and fills in the additional DATETIME fields with zeros. For example, 08/15/2002 becomes 2002-08-15 00:00:00.
- If you convert DATETIME YEAR TO SECOND to DATE, the database server converts existing DATETIME to DATE values but drops fields for time units smaller than DAY. For example, 2002-08-15 12:15:37 becomes 08/15/2002.

Using User-Defined Casts

Implicit and explicit casts are owned by the users who create them. They govern casts and conversions between user-defined data types and other data types. Developers of user-defined data types must create certain implicit and explicit casts and the functions that are used to implement them. The casts allow user-defined types to be expressed in a form that clients can manipulate.

For information about how to register and use implicit and explicit casts, see the CREATE CAST statement in the *HCL® Informix® Guide to SQL: Syntax* and the *IBM® Informix® Database Design and Implementation Guide*.

Implicit Casts

Implicit casts allow you to convert a user-defined data type to a built-in type or vice versa. The database server automatically invokes a single implicit cast when it must evaluate and compare expressions or pass arguments. Operations that require more than one implicit cast fail.

Users can explicitly invoke an implicit cast using the CAST AS keywords or the double colon (**::**) cast operator.

Explicit Casts

Explicit casts, unlike implicit casts or built-in casts, are *never* invoked automatically by the database server. Users must invoke them explicitly with the CAST AS keywords or with the double colon (**::**) cast operator.

Determining Which Cast to Apply

The database server uses the following rules to determine which cast to apply in a particular situation:

- To compare two built-in types, the database server automatically invokes the appropriate built-in casts.
- The database server applies only one implicit cast per operand. If two or more casts are required to convert the operand to the specified type, the user must explicitly invoke the additional casts.

  In the following example, the literal value `5.55` is implicitly cast to DECIMAL, and is then explicitly cast to MONEY, and finally to `yen`:

  ```
  CREATE DISTINCT TYPE yen AS MONEY
  . . .
  INSERT INTO currency_tab
     VALUES (5.55::MONEY::yen)
  ```

- To compare a distinct type to its source type, the user must explicitly cast one type to the other.
- To compare a distinct type to a type other than its source, the database server looks for an implicit cast between the source type and the specified type.

  If neither cast is registered, the user must invoke an explicit cast between the distinct type and the specified type. If this cast is not registered, the database server automatically invokes a cast from the source type to the specified type.

  If none of these casts is defined, the comparison fails.

- To compare an opaque type to a built-in type, the user must explicitly cast the opaque type to a data type that the database server understands (such as LVARCHAR, SENDRECV, IMPEXP, or IMPEXPBIN). The database server then invokes built-in casts to convert the results to the specified built-in type.
- To compare two opaque types, the user must explicitly cast one opaque type to a form that the database server understands (such as LVARCHAR, SENDRECV, IMPEXP, or IMPEXPBIN) and then explicitly cast this type to the second opaque type.

For information about casting and the BOOLEAN, BSON, JSON, IMPEXP, IMPEXPBIN, LVARCHAR, and SENDRECV built-in opaque data types, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

Casts for distinct types

You define a distinct type based on a built-in type or an existing opaque type or ROW type. Although data of the distinct type has the same length and alignment and is passed in the same way as data of the source type, the two cannot be compared directly. To compare a distinct type and its source type, you must explicitly cast one type to the other.

When you create a new distinct type, the database server automatically registers two explicit casts:

- A cast from the distinct type to its source type
- A cast from the source type to the distinct type

You can create an implicit cast between a distinct type and its source type. To create an implicit cast, however, you must first drop the default explicit cast between the distinct type and its source type.

You also can use all casts that have been registered for the source type without modification on the distinct type. You can also create and register new casts and support functions that apply only to the distinct type.

For examples that show how to create a cast function for a distinct type and register the function as cast, see the *IBM® Informix® Database Design and Implementation Guide*.

Important: For releases of HCL Informix® earlier than Version 9.21, distinct data types inherited the built-in casts that are provided for the source type. The built-in casts of the source type are not inherited by distinct data types in this release.

What Extended Data Types Can Be Cast?

The next table shows the extended data type combinations that you can cast.

Table 80. Extended data type combinations

Target Type	Opaque Type	Distinct Type	Named ROW Type	Unnamed ROW Type	Collection Type	Built-in Type
Opaque Type	Explicit or implicit	Explicit	Explicit	Not Valid	Not Valid	Explicit or implicit[3]
Distinct Type	Explicit[3]	Explicit	Explicit	Not Valid	Not Valid	Explicit or implicit
Named ROW Type	Explicit[3]	Explicit	Explicit[3]	Explicit[1]	Not Valid	Not Valid
Unnamed ROW Type	Not Valid	Not Valid	Explicit[1]	Implicit[1]	Not Valid	Not Valid
Collection Type	Not Valid	Not Valid	Not Valid	Not Valid	Explicit[2]	Not Valid

Table 80. Extended data type combinations (continued)

Target Type	Opaque Type	Distinct Type	Named ROW Type	Unnamed ROW Type	Collection Type	Built-in Type
Built-in Type	Explicit or implicit[3]	Explicit or implicit	Not Valid	Not Valid	Not Valid	System defined (implicit)

[1] Applies when two ROW types are structurally equivalent or casts exist to handle data conversions where corresponding field types are not the same.

[2] Applies when a cast exists to convert between the element types of the respective collection types.

[3] Applies when a user-defined cast exists to convert between the two data types.

The table shows only whether a cast between a source type and a target type are possible. In some cases, you must first create a user-defined cast before you can perform a conversion between two data types. In other cases, the database server provides either an implicit cast or a built-in cast that you must explicitly invoke.

Operator Precedence

An *operator* is a symbol or keyword that can be in an SQL expression. Most SQL operators are restricted in the data types of their operands and returned values. Some operators only support operands of built-in data types; others can support built-in and extended data types as operands.

The following table shows the precedence of the operators thatHCL Informix® supports, in descending (highest to lowest) order of precedence. Operators with the same precedence are listed in the same row.

Operator Precedence	Example in Expression
. (*membership*) **[ ]** (*substring*)	**customer.phone [**1, 3**]**
UNITS	**x** UNITS DAY
+ - (*unary*)	**- y**
:: (*cast*)	NULL**::**TEXT
*** /**	**x / y**
+ - (*binary*)	**x -y**
|| (*concatenation*)	**customer.fname** || **customer.lname**
ANY ALL SOME	**orders.ship_date** > SOME (SELECT **paid_date** FROM **orders**)
NOT	NOT **y**
< <= = > >= != <>	**x >= y**
IN BETWEEN ... AND LIKE MATCHES	**customer.fname** MATCHES **y**

Operator Precedence	Example in Expression
AND	**x** AND **y**
OR	**x** OR **y**

See the *HCL® Informix® Guide to SQL: Syntax* for the syntax and semantics of these SQL operators.

Environment variables

Various *environment variables* affect the functionality of your HCL Informix® products. You can set environment variables that identify your terminal, specify the location of your software and define other parameters.

Some environment variables are required; others are optional. You must either set or accept the default setting for required environment variables.

These topics describe how to use the environment variables that apply to one or more HCL Informix® products and shows how to set them.

Types of environment variables

Two types of environment variables are explained in this chapter:

- Environment variables that are specific to HCL Informix®

  Set HCL Informix® environment variables when you want to work with HCL Informix® products. Each HCL Informix® product publication specifies the environment variables that you must set to use that product.
- Environment variables that are used with a specific operating system

  HCL Informix® products rely on the correct setting of certain standard operating system environment variables. For example, you must always set the **PATH** environment variable.

In a UNIX™ environment, you might also be required to set the TERMCAP or TERMINFO environment variable to use some products effectively.

The GLS environment variables that support nondefault locales are described in the *HCL® Informix® GLS User's Guide*. The GLS variables are included in the list of environment variables in Table 81: Uses for environment variables on page 1449.

The database server uses the environment variables that were in effect at the time when the database server was initialized.

The onstat - g env command lists the active environment settings.

Tip: Additional environment variables that are specific to your client application or SQL API might be explained in the publication for that product.

Important: Do not set any environment variable in the home directory of user **informix** (or in the file `.informix` in that directory) while initializing the database and creating the **sysmaster** database.

Limitations on environment variables

Size of a block of environment variables

At the start of a session, the client groups all the environment variables that the server will use and sends the environment variables to the server as single block. The maximum size of this block is 32K. If the block of environment variables is greater than 32K, the error -1832 is returned to the application. The text of this error is "Environment block is greater than 32K."

To resolve this error, you can either unset one or more environment variables or reduce the size of some of the environment variables.

Using environment variables on UNIX™

You can set, unset, modify, and view environment variables. If you already use any HCL Informix® products, some or all of the appropriate environment variables might be set.

You can set environment variables on UNIX™ in the following places:

- At the system prompt on the command line

  When you set an environment variable at the system prompt, you must reassign it the next time you log in to the system.

- In an environment-configuration file

  An environment-configuration file is a common or private file where you can set all the environment variables that HCL Informix® products use. The use of such files reduces the number of environment variables that you must set at the command line or in a shell file.

- In a login file

  Values of environment variables set in your `.login`, `.cshrc`, or `.profile` file are assigned automatically every time you log in to the system.

- In the SET ENVIRONMENT statement of SQL

  Values of some environment variables can reset by the SET ENVIRONMENT statement. The scope of the new settings is generally the routine that executed the SET ENVIRONMENT statement, but it is the current session for the

OPTCOMPIND environment variable of Informix®, as described in the section OPTCOMPIND environment variable on page 1504. For more information about these routines and on the SET ENVIRONMENT statement, see the *HCL® Informix® Guide to SQL: Syntax*.

In , you can set supported environment variables within an application with the putenv() system call and retrieve values with the getenv() system call, if your UNIX™ system supports these functions. For more information about putenv() and getenv(), see the *HCL® Informix® Enterprise Replication Guide* and your C documentation.

Setting environment variables in a configuration file

About this task

The common (shared) environment-configuration file that is provided with HCL Informix® products is located in **$INFORMIXDIR/etc/informix.rc**. Permissions for this shared file must be set to `644`.

A user can override the system or shared environment variables by setting variables in a private environment-configuration file. This file must have all of the following characteristics:

- Stored in the user's home directory
- Named **.informix**
- Permissions set to readable by the user

An environment-configuration file can contain comment lines (preceded by the `#` comment indicator) and variable definition lines that set values (separated by blank spaces or tabs), as the following example shows:

```
# This is an example of an environment-configuration file
#
DBDATE DMY4-
#
# These are ESQL/C environment variable settings
#
INFORMIXC gcc
CPFIRST TRUE
```

You can use the **ENVIGNORE** environment variable, described in ENVIGNORE environment variable (UNIX) on page 1482, to override one or more entries in an environment-configuration file. Use the HCL Informix® **chkenv** utility, described in Checking environment variables with the chkenv utility on page 1444, to perform a validity check on the contents of an environment-configuration file. The **chkenv** utility returns an error message if the file contains a bad environment variable or if the file is too large.

The first time you set an environment variable in a shell file or environment-configuration file, you must tell the shell process to read your entry before you work with your HCL Informix® product. If you use a C shell, **source** the file; if you use a Bourne or Korn shell, use a period (**.**) to execute the file.

Setting environment variables at login time

About this task

Add commands that set your environment variables to the appropriate login file:

For C shell

.login or **.cshrc**

For Bourne shell or Korn shell

.profile

Syntax for setting environment variables

Use standard UNIX™ commands to set environment variables. The examples in the following table show how to set the ABCD environment variable to *value* for the C shell, Bourne shell, and Korn shell. The Korn shell also supports a shortcut, as the last row indicates. Environment variables are case-sensitive.

Shell	Command
C	`setenv ABCD value`
Bourne	`ABCD=value` `export ABCD`
Korn	`ABCD=value` `export ABCD`
Korn	`export ABCD=value`

The following diagram shows how the syntax for setting an environment variable is represented throughout this chapter. These diagrams indicate the setting for the C shell; for the Bourne or Korn shells, use the syntax illustrated in the preceding table.



Unsetting environment variables

About this task

To unset an environment variable, enter the following command.

Shell	Command
C	`unsetenv ABCD`
Bourne or Korn	`unset ABCD`

Modifying an environment-variable setting

About this task

Sometimes you must add information to an environment variable that is already set. For example, the **PATH** environment variable is always set on UNIX™. When you use HCL Informix® productd, you must add to the **PATH** setting the name of the directory where the executable files for the HCL Informix® products are stored.

In the following example, the **INFORMIXDIR** is **/usr/informix**. (That is, during installation, the HCL Informix® products were installed in the **/usr /informix** directory.) The executable files are in the **bin** subdirectory, **/usr/informix/bin**. To add this directory to the front of the C shell **PATH** environment variable, use the following command:

```
setenv PATH /usr/informix/bin:$PATH
```

Rather than entering an explicit pathname, you can use the value of the **INFORMIXDIR** environment variable (represented as **$INFORMIXDIR**), as the following example shows:

```
setenv INFORMIXDIR /usr/informix
setenv PATH $INFORMIXDIR/bin:$PATH
```

You might prefer to use this version to ensure that your **PATH** entry does not conflict with the search path that was set in **INFORMIXDIR**, and so that you are not required to reset **PATH** whenever you change **INFORMIXDIR**. If you set the **PATH** environment variable on the C shell command line, you might be required to include braces (`{}`) with the existing **INFORMIXDIR** and **PATH**, as the following command shows:

```
setenv PATH ${INFORMIXDIR}/bin:${PATH}
```

For more information about how to set and modify environment variables, see the publications for your operating system.

Viewing your environment-variable settings

About this task

After you install one or more HCL Informix® products, enter the following command at the system prompt to view your current environment settings.

UNIX™ version	Command
BSD UNIX™	env
UNIX™ System V	printenv

Checking environment variables with the chkenv utility

About this task

The **chkenv** utility checks the validity of shared or private environment-configuration files. It validates the names of the environment variables in the file, but not their values. Use **chkenv** to provide debugging information when you define, in an environment-configuration file, all the environment variables that your HCL Informix® products use.

chkenv *[pathname] filename*

filename

is the name of the environment-configuration file to be debugged.

pathname

is the full directory path in which the environment variable file is located.

File **$INFORMIXDIR/etc/informix.rc** is the shared environment-configuration file. A private environment-configuration file is stored as **.informix** in the home directory of the user. If you specify no *pathname* for **chkenv**, the utility checks both the shared and private environment configuration files. If you provide a pathname, **chkenv** checks only the specified file.

Issue the following command to check the contents of the shared environment-configuration file:

```
chkenv informix.rc
```

The **chkenv** utility returns an error message if it finds a bad environment-variable name in the file or if the file is too large. You can modify the file and rerun the utility to check the modified environment-variable names.

HCL Informix® products ignore all lines in the environment-configuration file, starting at the point of the error, if the **chkenv** utility returns the following message:

```
-33523     filename: Bad environment variable on line number.
```

If you want the product to ignore specified environment-variables in the file, you can also set the **ENVIGNORE** environment variable. For a discussion of the use and format of environment-configuration files and the **ENVIGNORE** environment variable, see page ENVIGNORE environment variable (UNIX) on page 1482.

Rules of precedence for environment variables

When HCL Informix® products accesses an environment variable, normally the following rules of precedence apply:

1. Of highest precedence is the value that is defined in the environment (shell) by explicitly setting the value at the shell prompt.
2. The second highest precedence goes to the value that is defined in the private environment-configuration file in the home directory of the user (**~/.informix**).
3. The next highest precedence goes to the value that is defined in the common environment-configuration file ($**INFORMIXDIR/etc/informix.rc**).
4. The lowest precedence goes to the default value, if one exists.

For precedence information about GLS environment variables, see the *HCL® Informix® GLS User's Guide*.

Important: If you set one or more environment variables before you start the database server, and you do not explicitly set the same environment variables for your client products, the clients will adopt the original settings.

Using environment variables on Windows™

The following sections discuss setting, viewing, unsetting, and modifying environment variables for Windows™ applications.

Where to set environment variables on Windows™

You can set environment variables in several places on Windows™, depending on which HCL Informix® application you use.

Environment variables can be set in several ways, as described in Setting environment variables on Windows on page 1446.

The SET ENVIRONMENT statement of SQL can set certain routine-specific environment options. For more information, see the description of SET ENVIRONMENT in the *HCL® Informix® Guide to SQL: Syntax*.

To use client applications such as or the Schema Tools on Windows™ environment, use the Setnet32 utility to set environment variables. For information about the Setnet32 utility, see the *Informix® Client Products Installation Guide* for your operating system.

In , you can set supported environment variables within an application with the ifx_putenv() function and retrieve values with the ifx_getenv() function, if your Windows™ system supports them. For more information about ifx_putenv() and ifx_getenv(), see the *HCL® Informix® Enterprise Replication Guide*.

Setting environment variables on Windows™

About this task

You can set environment variables for command-prompt utilities in the following ways:

- With the System applet in the Control Panel
- In a command-line session

Using the system applet to change environment variables

The System applet provides a graphical interface to create, modify, and delete system-wide and user-specific variables. Environment variables that are set with the System applet are visible to all command-prompt sessions.

About this task

To change environment variables with the System applet in the control panel

1. Double-click the System applet icon from the Control Panel window.
2. Click the Environment tab near the top of the window.

   Two list boxes display System Environment Variables and User Environment Variables. System Environment Variables apply to an entire system, and User Environment Variables apply only to the sessions of the individual user.
3. To change the value of an existing variable, select that variable. The name of the variable and its current value are in the boxes at the bottom of the window.
4. To add a new variable, highlight an existing variable and type the new variable name in the box at the bottom of the window.
5. Next, enter the value for the new variable at the bottom of the window and click **Set** .
6. To delete a variable, select the variable and click **Delete**.

Results

Important: In order to use the System applet to change System environment variables, you must belong to the Administrators group. For information about assigning users to groups, see your operating-system documentation.

Using the command prompt to change environment variables

You can change the setting of an environment variable at a command prompt.

About this task

The following diagram shows the syntax for setting an environment variable at a command prompt in Windows™.

```
setABCD=value
```

If no *value* is specified, the environment variable is unset, as if it did not exist.

To view your current settings after one or more HCL Informix® products are installed, enter the following command at the command prompt.

```
set
```

Sometimes you must add information to an environment variable that is already set. For example, the **PATH** environment variable is always set in Windows™ environments. When you use HCL Informix® products, you must add the name of the directory where the executable files for the HCL Informix® products are stored to the **PATH**.

In the following example, **INFORMIXDIR** is `d:\informix` (that is, during installation, HCL Informix® products were installed in the `d: \informix` directory). The executable files are in the `bin` subdirectory, `d:\informix\bin`. To add this directory at the beginning of the **PATH** environment-variable value, use the following command:

```
set PATH=d:\informix\bin;%PATH%
```

Rather than entering an explicit pathname, you can use the value of the **INFORMIXDIR** environment variable (represented as **%INFORMIXDIR%**), as the following example shows:

```
set INFORMIXDIR=d:\informix
set PATH=%PATH%
```

You might prefer to use this version to ensure that your **PATH** entry does not contradict the search path that was set in **INFORMIXDIR** and to avoid the requirement to reset **PATH** whenever you change **INFORMIXDIR**.

For more information about setting and modifying environment variables, see your operating-system publications.

Using dbservername.cmd to initialize a command-prompt environment

Each time that you open a Windows™ command prompt, it acts as an independent environment. Therefore, environment variables that you set within it are valid only for that particular command-prompt instance.

About this task

For example, if you open one command window and set the variable, **INFORMIXDIR**, and then open another command window and type `set` to check your environment, you will find that **INFORMIXDIR** is not set in the new command-prompt session.

The database server installation program creates a batch file that you can use to configure command-prompt utilities, ensuring that your command-prompt environment is initialized correctly each time that you run a command-prompt session. The batch file, `dbservername.cmd`, is located in `%INFORMIXDIR%`, and is a plain text file that you can modify with any text editor. If you have more than one database server installed in `%INFORMIXDIR%`, there will be more than one batch file with the `.cmd` extension, each bearing the name of the database server with which it is associated.

To run `dbservername.cmd` from a command prompt, type `dbservername` or configure a command prompt so that it runs `dbservername.cmd` automatically at start.

Rules of precedence for Windows™ environment variables

When HCL Informix® products access an environment variable, normally the following rules of precedence apply:

1. The setting in Setnet32 with the **Use my settings** box selected.
2. The setting in Setnet32 with the **Use my settings** box cleared.
3. The setting on the command line before running the application.
4. The setting in Windows™ as a user variable.
5. The setting in Windows™ as a system variable.
6. The lowest precedence goes to the default value.

An application examines the first five values as it starts. Unless otherwise stated, changing an environment variable after the application is running does not have any effect.

Environment variables in Informix® products

The topics that follow discuss (in alphabetic order) environment variables that HCL Informix® database server products and their utilities use.

Important: The descriptions of the following environment variables include the syntax for setting the environment variable on UNIX™. For a general description of how to set these environment variables on Windows™, see Setting environment variables on Windows on page 1446.

Related information

Informix environment variables with the IBMInformix JDBC Driver on page

GLS-related environment variables on page

Enterprise Replication configuration parameter and environment variable reference on page

AC_CONFIG file environment variable on page

Environment variable portal

This portal is an index of usage categories for HCL Informix® and UNIX™ environment variables. The portal contains links to the topics that describe the environment variables.

Because the following table contains a comprehensive list of categories with links to applicable topics. Some environment variables are applicable for more than one category.

Table 81. Uses for environment variables

Functional category	Environment variable
Abbreviated year values	Specify how to expand literal DATE and DATETIME values: DBCENTURY environment variable on page 1467
ANSI/ISO SQL compliance	Set the case of owner names: ANSIOWNER environment variable on page 1461 Specify if you want to check for HCL Informix® extensions to ANSI-standard SQL syntax: DBANSIWARN environment variable on page 1466 No default table or routine access privileges for PUBLIC in databases not created WITH LOG MODE ANSI: NODEFDAC environment variable on page 1503
archecker utility	Specify the full path name for the archecker configuration file: AC_CONFIG file environment variable on page
Buffers	Manage the fetch buffer size: FET_BUF_SIZE environment variable on page 1483 Manage the network size: IFX_NETBUF_SIZE environment variable on page 1489 Manage the network pool size: IFX_NETBUF_PVTPOOL_SIZE environment variable (UNIX) on page 1488 Manage the BYTE or TEXT data buffer: DBBLOBBUF environment variable on page 1466
Cache	Control the use of the shared-statement cache on a session: STMT_CACHE environment variable on page 1513
Client/server	Specify the default database server: INFORMIXSERVER environment variable on page 1498

Table 81. Uses for environment variables (continued)

Functional category	Environment variable
	Specify where shared-memory segments are attached to the client process: INFORMIXSHMBASE environment variable (UNIX) on page 1498 Specify the stack size for a client process: INFORMIXSTACKSIZE environment variable on page 1499 Specify locale information, including for the client and server: GLS-related environment variables on page
Code-set conversion	Specify locale and multibyte information: GLS-related environment variables on page Specify the location of the **concsm.cfg** file: INFORMIXCONCSMCFG environment variable on page 1494 Specify the filename or pathname of the C compiler: INFORMIXC environment variable (UNIX) on page 1493 Specify the pathname of the map file for C++ programs: INFORMIXCPPMAP environment variable on page 1497 Specify information for compiling multithreaded applications: THREADLIB environment variable (UNIX) on page 1515
Configuration	Specify the name of the active that holds configuration parameters: ONCONFIG environment variable on page 1503 Ignore specified environment variable settings: ENVIGNORE environment variable (UNIX) on page 1482 Specify the default database server: INFORMIXSERVER environment variable on page 1498 Specify the dbspaces in which temporary tables are built: DBSPACETEMP environment variable on page 1477 Manage query optimizer directives: IFX_DIRECTIVES environment variable on page 1485 and IFX_EXTDIRECTIVES environment variable on page 1485

Table 81. Uses for environment variables (continued)

Functional category	Environment variable
	Modify the value of the OPTCOMPIND configuration parameter: OPTCOMPIND environment variable on page 1504 Specify the query performance goal for the optimizer: OPT_GOAL environment variable (UNIX) on page 1506 Specify the degree of parallelism that the database server uses: PDQPRIORITY environment variable on page 1507 Specify the stack size that is applied to all client processes: INFORMIXSTACKSIZE environment variable on page 1499
Connecting	Set the maximum number of *additional* connection attempts: INFORMIXCONRETRY environment variable on page 1495 Set connect time information: INFORMIXCONTIME environment variable on page 1496 Specify the default database server to for connections: INFORMIXSERVER environment variable on page 1498 Specify the location of connection information: INFORMIXSQLHOSTS environment variable on page 1499
Connection Manager	Specify the location of the Connection Manager configuration file:CMCONFIG environment variable on page 1462
Data distributions	Manage the amount of system disk space that the UPDATE STATISTICS statement can use: DBUPSPACE environment variable on page 1479
Database locale	Manage locale information: GLS-related environment variables on page
Database server	Specify servers for connections: INFORMIXSERVER environment variable on page 1498 Set the locale for file I/O: GLS-related environment variables on page Specify the name of the active file that holds configuration parameters: ONCONFIG environment variable on page 1503

Table 81. Uses for environment variables (continued)

Functional category	Environment variable
	Manage parallel sorting: PSORT_DBTEMP environment variable on page 1510 and PSORT_NPROCS environment variable on page 1511 Manage parallelism: PDQPRIORITY environment variable on page 1507 Manage role separation: INF_ROLE_SEP environment variable on page 1500 Manage shared memory: INFORMIXSHMBASE environment variable (UNIX) on page 1498 Manage stack size: INFORMIXSTACKSIZE environment variable on page 1499 Manage temporary tables: DBSPACETEMP environment variable on page 1477, DBTEMP environment variable on page 1478, and PSORT_DBTEMP environment variable on page 1510 Manage variable-length packets: IFX_PAD_VARCHAR environment variable on page 1491
Date and time values, formats	Manage date and time information: DBCENTURY environment variable on page 1467, DBDATE environment variable on page 1469, #unique_1978, GLS-related environment variables on page (**GL_DATE** and **GL_DATETIME**), The USE_DTENV environment variable on page , and TZ environment variable on page 1515
DB-Access utility	Manage the database server and DB-Access: DBANSIWARN environment variable on page 1466, DBDELIMITER environment variable on page 1472, DBEDIT environment variable on page 1472. DBFLTMASK environment variable on page 1472, DBPATH environment variable on page 1474, FET_BUF_SIZE environment variable on page 1483, INFORMIXSERVER environment variable on page 1498, INFORMIXTERM environment variable (UNIX) on page 1500, TERM environment variable (UNIX) on page 1514, TERMCAP environment variable (UNIX) on

Table 81. Uses for environment variables (continued)

Functional category	Environment variable
	page 1514, and TERMINFO environment variable (UNIX) on page 1515
dbexport utility	Set the field delimiter: DBDELIMITER environment variable on page 1472
dbinfo	Specify if dbinfo('dbspace', partnum) raises error -727 or returns NULL for an invalid partnum: DBINFO_DBSPACE_RETURN_NULL_FOR_INVALID_PARTNUM environment variable on page 1473
Delimited identifiers	Set the field delimiter used with the dbexport utility and with the LOAD and UNLOAD statements: DBDELIMITER environment variable on page 1472
Disk space	Manage the amount of system disk space and memory that the UPDATE STATISTICS MEDIUM or HIGH statement can use: DBUPSPACE environment variable on page 1479
Editor	Specify the text editor to use with SQL statements and command files in DB-Access: DBEDIT environment variable on page 1472
Enterprise Replication	Specify information for Enterprise Replication: configuration parameter and environment variable reference on page
ESQL/C	Specify ANSI compliance: DBANSIWARN environment variable on page 1466 Specify the filename or pathname of the C compiler to use with ESQL/C: INFORMIXC environment variable (UNIX) on page 1493 Set delimited identifiers: DELIMIDENT environment variable on page 1481 Specify multibyte characters and locale information GLS-related environment variables on page (**CLIENT_LOCALE**, **ESQLMF**, and **GL_USER**) Specify information for multithreaded applications: THREADLIB environment variable (UNIX) on page 1515

Table 81. Uses for environment variables (continued)

Functional category	Environment variable
	Specify the default compilation order: CPFIRST environment variable on page 1461
Executable programs	Specify the directories to search for executable programs: PATH environment variable on page 1507
Fetch buffer size	Set buffer size information: FET_BUF_SIZE environment variable on page 1483
Filenames: multibyte	GLS-related environment variables on page (**GLS8BITFSYS**)
Files: field delimiter	Set the field delimiter: DBDELIMITER environment variable on page 1472
Files: installation	Specify the directory that contains the subdirectories in which your product files are installed: INFORMIXDIR environment variable on page 1497
Files: locale	Specify locale information: GLS-related environment variables on page (**CLIENT_LOCALE**, **DB_LOCALE**, and **SERVER_LOCALE**)
Files: map for C++	Specify the pathname of the map file for C++ programs: INFORMIXCPPMAP environment variable on page 1497
Files: message	Specify the subdirectory of **$INFORMIXDIR** or the pathname of the directory that contains the compiled message files that the database server uses: DBLANG environment variable on page 1473
Files: temporary	DBSPACETEMP environment variable on page 1477
Files: temporary	Specify a directory for temporary files: DBTEMP environment variable on page 1478
Files: temporary sorting	Specify the location of temporary files used for sorting: PSORT_DBTEMP environment variable on page 1510
Files: `termcap`, `terminfo`	Specify terminal information: INFORMIXTERM environment variable (UNIX) on page 1500, TERM environment variable (UNIX) on page 1514, TERMCAP environment variable (UNIX) on page 1514, and TERMINFO environment variable (UNIX) on page 1515

Table 81. Uses for environment variables (continued)

Functional category	Environment variable
Format: date and time	Define the format for date and time information: DBCENTURY environment variable on page 1467, DBDATE environment variable on page 1469, #unique_1978, GLS-related environment variables on page (**GL_DATE** and **GL_DATETIME**), The USE_DTENV environment variable on page , and TZ environment variable on page 1515
Format: private-use characters	Set the display width for characters in Unicode Private-Use Area (PUA) ranges: GLS-related environment variables on page (IFX_PUA_DISPLAY_MAPPING)
Format: money	Define the format for money information: #unique_1962and GLS-related environment variables on page
Identifiers	Specify field delimiters: DELIMIDENT environment variable on page 1481 Specify information for identifiers longer than 18 bytes: IFX_LONGID environment variable on page 1488 Specify information for multibyte characters: GLS-related environment variables on page (**CLIENT_LOCALE** and **ESQLMF**)
Installation	Specify the directory that contains the subdirectories in which your product files are installed: INFORMIXDIR environment variable on page 1497 Specify which directories to search for executable programs: PATH environment variable on page 1507
JDBC	Manage environment variables used with JDBC: Informix® environment variables with the HCL Informix® JDBC Driver on page
Language environment	Specify language and locale information: DBLANG environment variable on page 1473 and GLS-related environment variables on page
Libraries	Specify paths for libraries: LD_LIBRARY_PATH environment variable (UNIX) on page 1502, LIBPATH environment variable (UNIX) on page 1503, and #unique_2070

Table 81. Uses for environment variables (continued)

Functional category	Environment variable
Locale	Define client, server, and database locale information: GLS-related environment variables on page (**CLIENT_LOCALE**, **DB_LOCALE**, and **SERVER_LOCALE**)
Lock mode	Set the default lock mode for database tables that are created without specifying the LOCKMODE PAGE or LOCKMODE ROW keywords: IFX_DEF_TABLE_LOCKMODE environment variable on page 1484
Long Identifiers	Specify information for identifiers longer than 18 bytes: IFX_LONGID environment variable on page 1488
Map file for C++	Specify the pathname of the map file for C++ programs: INFORMIXCPPMAP environment variable on page 1497
Message chaining	Enable or disable optimized message transfers (message chaining) for : OPTMSG environment variable on page 1505
Message files	Specify the directory that contains compiled message files: DBLANG environment variable on page 1473
Money format	Define the format for money information: #unique_1962and GLS-related environment variables on page
Multibyte characters	Specify information for multibyte characters: GLS-related environment variables on page (**CLIENT_LOCALE**, **DB_LOCALE**, **SERVER_LOCALE**, and **GL_USEGLU**)
Multibyte filter	Specify Informix® ESQL/C multibyte filter information: GLS-related environment variables on page (**ESQLMF**)
Multithreaded applications	Specify information for compiling multithreaded applications: THREADLIB environment variable (UNIX) on page 1515
Network	Specify network information: DBPATH environment variable on page 1474
Nondefault locale	Define client, server, and database locale information: GLS-related environment variables on page (**CLIENT_LOCALE**, **DB_LOCALE**, and **SERVER_LOCALE**)
utility	Optimize the deduplication capabilities for storage managers: IFX_BAR_USE_DEDUP environment variable on page

Table 81. Uses for environment variables (continued)

Functional category	Environment variable
	Disable the ability to replicate, import, or export backup objects among TSM servers: IFX_TSM_OBJINFO_OFF environment variable on page
ONCONFIG parameters	Specify the name of the file that holds configuration parameters: ONCONFIG environment variable on page 1503
oninit output (Windows™ only)	Specify a path and file for oninit output: ONINIT_STDOUT environment variable (Windows) on page 1504
Optimization: directives	Manage query optimizer directives: IFX_DIRECTIVES environment variable on page 1485 and IFX_EXTDIRECTIVES environment variable on page 1485
Optimization: message transfers	Enable or disable optimized message transfers (message chaining) for : OPTMSG environment variable on page 1505
Optimization: join method	Modify the value of the OPTCOMPIND configuration parameter: OPTCOMPIND environment variable on page 1504
Optimization: performance goal	Specify the query performance goal for the optimizer: OPT_GOAL environment variable (UNIX) on page 1506
OPTOFC feature	Enable optimize-OPEN-FETCH-CLOSE functionality: OPTOFC environment variable on page 1506
PAM authentication for MongoDB clients	Enable PAM authentication for MongoDB clients: IFMXMONGOAUTH environment variable on page 1483
Path name: **archecker** configuration file	Specify the full path name for the archecker configuration file: AC_CONFIG file environment variable on page
Path name: C compiler	Specify the filename or pathname of the C compiler: INFORMIXC environment variable (UNIX) on page 1493
Path name: database files	Specify database server file and path information: DBPATH environment variable on page 1474
Path name: executable programs	Specify directories to search for executable programs: PATH environment variable on page 1507
Path name: installation	Specify the directory that contains the subdirectories in which your product files are installed: INFORMIXDIR environment variable on page 1497

Table 81. Uses for environment variables (continued)

Functional category	Environment variable
Path name: libraries	Specify paths for libraries: LD_LIBRARY_PATH environment variable (UNIX) on page 1502, LIBPATH environment variable (UNIX) on page 1503, and #unique_2070
Path name: message files	Specify the directory that contains compiled message files: DBLANG environment variable on page 1473 and GLS-related environment variables on page
Path name: parallel sorting	Specify the location of temporary files for sorts: PSORT_DBTEMP environment variable on page 1510
HCL Informix® Primary Storage Manager	Manage the storage manager: PSM_ACT_LOG environment variable on page 1508. PSM_CATALOG_PATH environment variable on page 1508, PSM_DBS_POOL environment variable on page 1509, PSM_DEBUG environment variable on page 1509, PSM_DEBUG_LOG environment variable on page 1510, and PSM_LOG_POOL environment variable on page 1510
Preserve owner name case	Set the case of owner names: ANSIOWNER environment variable on page 1461
Printing	Specify the default printing program: DBPRINT environment variable on page 1476
Privileges	Configure role separation: INF_ROLE_SEP environment variable on page 1500
Query: optimization	Manage query optimizer directives: IFX_DIRECTIVES environment variable on page 1485 and IFX_EXTDIRECTIVES environment variable on page 1485 Modify the value of the OPTCOMPIND configuration parameter: OPTCOMPIND environment variable on page 1504 Specify the query performance goal for the optimizer: OPT_GOAL environment variable (UNIX) on page 1506 Specify user-defined data types can use to estimate the cost of an R-tree index for queries on UDT columnsRTREE_COST_ADJUST_VALUE environment variable on page 1512

Table 81. Uses for environment variables (continued)

Functional category	Environment variable
Query: prioritization	Specify the degree of parallelism that the database server uses: PDQPRIORITY environment variable on page 1507
Remote shell	Specify information that overrides the default remote shell for performing remote tape operations: DBREMOTECMD environment variable (UNIX) on page 1477
Role separation	Configure role separation: INF_ROLE_SEP environment variable on page 1500
Rolled-back transactions	Manage what the DB-Access utility does when an error occurs: DBACCNOIGN environment variable on page 1465 Specify whether an internal rollback of a global transaction frees the transaction: IFX_XASTDCOMPLIANCE_XAEND environment variable on page 1492
Server locale	Define the locale of your database server: GLS-related environment variables on page **SERVER_LOCALE**
Shared memory	Specify where shared-memory segments are attached to the client process: INFORMIXSHMBASE environment variable (UNIX) on page 1498
Shell: remote	Specify information that overrides the default remote shell for performing remote tape operations: DBREMOTECMD environment variable (UNIX) on page 1477
Shell: search path	Specify which directories to search for executable programs: PATH environment variable on page 1507
Sorting	Specify the location of temporary files for sorts: PSORT_DBTEMP environment variable on page 1510 Allocate more threads for sorting: PSORT_NPROCS environment variable on page 1511
SQL statements	Specify information for caching: STMT_CACHE environment variable on page 1513 Specify connection information: INFORMIXCONRETRY environment variable on page 1495, INFORMIXCONTIME environment variable on page 1496, and INFORMIXSERVER environment variable on page 1498

Table 81. Uses for environment variables (continued)

Functional category	Environment variable
	Specify information for CREATE TEMP TABLE operations: DBSPACETEMP environment variable on page 1477 Specify information for DESCRIBE FOR UPDATE operations: IFX_UPDDESC environment variable on page 1491 Specify information for LOAD and UNLOAD operations: DBDELIMITER environment variable on page 1472 and DBBLOBBUF environment variable on page 1466 Specify information for SELECT INTO TEMP operations: DBSPACETEMP environment variable on page 1477 Specify information for SET PDQPRIORITY operations: PDQPRIORITY environment variable on page 1507 Specify information for SET STMT_CACHE operations Specify information for UPDATE STATISTICS operations: DBUPSPACE environment variable on page 1479
Stack size	Define the stack size that is applied to client processes: INFORMIXSTACKSIZE environment variable on page 1499
Temporary tables	Define information for temporary tables: DBSPACETEMP environment variable on page 1477, DBTEMP environment variable on page 1478, and PSORT_DBTEMP environment variable on page 1510
Terminal handling	Specify terminal information: INFORMIXTERM environment variable (UNIX) on page 1500, TERM environment variable (UNIX) on page 1514, TERMCAP environment variable (UNIX) on page 1514, and TERMINFO environment variable (UNIX) on page 1515
Time-limited software license	Set information for trial or evaluation software warning messages: IFX_NO_TIMELIMIT_WARNING environment variable on page 1490
Variables: overriding	Deactivate some specified environment variable settings: ENVIGNORE environment variable (UNIX) on page 1482
Virtual memory segments on large pages	Specify whether the database server can use large pages on platforms where the hardware and the operating system

Table 81. Uses for environment variables (continued)

Functional category	Environment variable
	support large pages of shared memory: IFX_LARGE_PAGES environment variable on page 1486
Year values (abbreviated)	Specify how to expand DATE and DATETIME values that are entered as abbreviated year values: DBCENTURY environment variable on page 1467

ANSIOWNER environment variable

In an ANSI-compliant database, you can prevent the default behavior of upshifting lowercase letters in owner names that are not delimited by quotation marks by setting the **ANSIOWNER** environment variable to 1.

```
setenvANSIOWNER 1
```

To prevent upshifting of lowercase letters in owner names in an ANSI-compliant database, you must set **ANSIOWNER** before you initialize HCL Informix®.

The following table shows how an ANSI-compliant database of HCL Informix® stores or reads the specified name of a database object called **oblong** if you were the owner of **oblong** and your **userid** (in all lowercase letters) were **owen**:

Table 82. Lettercase of implicit, unquoted, and quoted owner names, with and without ANSIOWNER

Owner Format	Specification	ANSIOWNER = 1	ANSIOWNER Not Set
Implicit:	oblong	owen.oblong	OWEN.oblong
Unquoted:	owen.oblong	owen.oblong	OWEN.oblong
Quoted:	'owen'.oblong	owen.oblong	owen.oblong

Because they do not match the lettercase of your **userid**, any SQL statements that specified the formats that are stored as **OWEN.oblong** would fail with errors.

CPFIRST environment variable

Use the **CPFIRST** environment variable to specify the default compilation order for all source files in your programming environment.

```
setenvCPFIRST { TRUE | FALSE }
```

When you compile programs with **CPFIRST** not set, the preprocessor runs first, by default, on the program source file and then passes the resulting file to the C language preprocessor and compiler. You can, however, compile the program source file in the following order:

1. Run the C preprocessor
2. Run the preprocessor
3. Run the C compiler and linker

To use a nondefault compilation order for a specific program, you can either give the program source file a `.ecp` extension, run the -cp option with the **esql** command on a program source file with a `.ec` extension, or set **CPFIRST**.

Set **CPFIRST** to `TRUE` (uppercase only) to run the C preprocessor before the preprocessor on all source files in your environment, irrespective of whether the -cp option is passed to the **esql** command or the source files have the `.ec` or the `.ecp` extension.

To restore the default order on a system where the **CPFIRST** environment variable has been set to `TRUE`, you can set **CPFIRST** to `FALSE`. On UNIX™ systems that support the C shell, the following command has the same effect:

```
unsetenv CPFIRST
```

CMCONFIG environment variable

Set the **CMCONFIG** environment variable to specify the location of the Connection Manager configuration file. You use the configuration file to specify service level agreements and other Connection Manager configuration options.

setenvCMCONFIG*path/file_name*

path/file_name

is the full path and file name of a Connection Manager configuration file.

If the CMCONFIG environment variable is not set and the configuration file name is not specified on the oncmsm utility command line, the Connection Manager attempts to load the file from the following path and file name:

```
$INFORMIXDIR/etc/cmsm.cfg
```

Example

Examples

Suppose the CMCONFIG environment variable points to a valid path and file name of a Connection Manager configuration file. To reload a Connection Manager instance using the configuration file specified in the shell environment enter the following command:

```
./oncmsm -r
```

To shut down a Connection Manager instance using the configuration file specified in the shell environment:

```
./oncmsm -k
```

Related information

The oncmsm utility on page

Example of configuring connection management for a high-availability cluster on page

CLIENT_LABEL environment variable

Set the **CLIENT_LABEL** environment variable in CSDK 4.10.xC10 or JDBC 4.10.JC10 client to assign a character string to CSDK or JDBC client session and identify that character string on the database server. You use this for environments where same userid runs multiple instances of the same application, and there is a need to distinguish one session from the other.

```
onstat -g env sesID
```

```
select * from sysenvses where envses_name = CLIENT_LABEL
```

Example

CSDK Example

Suppose the CLIENT_LABEL is set to two different strings and the same esqlc program is executed with the session ids being 43 and 201:

```
bash-3.2$ export CLIENT_LABEL='csdk_client1'
bash-3.2$ ./myesqlc

bash-3.2$ export CLIENT_LABEL='csdk_client2'
bash-3.2$ ./myesqlc
```

```
onstat
```

```
onstat -g env 43

Informix Dynamic Server Version 14.10                                    -- On-Line -- Up 5 days 23:01:39 --
 210712 Kbytes

Environment for session 43:

Variable            Value [values-list]
CLIENT_LABEL        cdsk_client2
CLIENT_LOCALE       en_US.8859-1
CLNT_PAM_CAPABLE    1
⇓σνιπ◊

onstat -g env 201

Informix Dynamic Server Version 14.10                                   -- On-Line -- Up 5 days 23:02:41 -- 210712
 Kbytes

Environment for session 201:

Variable            Value [values-list]
CLIENT_LABEL        cdsk_client1
```

```
CLIENT_LOCALE       en_US.8859-1
CLNT_PAM_CAPABLE    1
```

sysmaster

```
select * from sysenvses where envses_name = 'CLIENT_LABEL'

envses_sid    201
envses_id     9
envses_name   CLIENT_LABEL
envses_value  cdsk_client1

envses_sid    43
envses_id     9
envses_name   CLIENT_LABEL
envses_value  cdsk_client2

2 row(s) retrieved.


Database closed.
```

Example

JDBC Example

Suppose the CLIENT_LABEL is set to two different strings in the JDBC connection URL and the same JDBC program is executed with the session ids being 232 and 234:

```
java myjdbc "jdbc:informix-sqli://myhost:52220:user=myuser;password=mypasswd;CLIENT_LABEL=jdbc_client1"

java myjdbc "jdbc:informix-sqli://myhost:52220:user=myuser;password=mypasswd;CLIENT_LABEL=jdbc_client2"
```

onstat

```
onstat -g env 232
IBM Informix Dynamic Server Version 12.10.FC10                              -- On-Line -- Up 6 days 00:56:26
 -- 210712 Kbytes

Environment for session 232:

Variable            Value [values-list]
CLIENT_LABEL        jdbc_client1
CLIENT_LOCALE       en_US.8859-1
CLNT_PAM_CAPABLE    1

onstat -g env 234

IBM Informix Dynamic Server Version 12.10.FC10                              -- On-Line -- Up 6 days 00:56:59
 -- 210712 Kbytes

Environment for session 234:

Variable            Value [values-list]
CLIENT_LABEL        jdbc_client2
CLIENT_LOCALE       en_US.8859-1
CLNT_PAM_CAPABLE    1
```

```
sysmaster
```

```
Database selected.

select * from sysenvses where envses_name = 'CLIENT_LABEL'

envses_sid    234
envses_id     9
envses_name   CLIENT_LABEL
envses_value  jdbc_client2

envses_sid    232
envses_id     9
envses_name   CLIENT_LABEL
envses_value  jdbc_client1

2 row(s) retrieved.

Database closed.
```

DBACCNOIGN environment variable

Use the **DBACCNOIGN** environment variable to specify the behavior of the DB-Access utility when specified errors occurs.

The **DBACCNOIGN** environment variable affects the behavior of the DB-Access utility if an error occurs under one of the following circumstances:

- You run DB-Access in non-menu mode.
- In HCL Informix® only, you execute the LOAD command with DB-Access in menu mode.

Set the **DBACCNOIGN** environment variable to `1` to roll back an incomplete transaction if an error occurs while you run the DB-Access utility under either of the preceding conditions.

```
setenvDBACCNOIGN1
```

For example, assume DB-Access runs the following SQL commands:

```
DATABASE mystore
BEGIN WORK

INSERT INTO receipts VALUES (cust1, 10)
INSERT INTO receipt VALUES (cust1, 20)
INSERT INTO receipts VALUES (cust1, 30)

UPDATE customer
   SET balance =
      (SELECT (balance-60)
      FROM customer WHERE custid = 'cust1')
   WHERE custid = 'cust1
COMMIT WORK
```

Here, one statement has a misspelled table name: the **receipt** table does not exist. If **DBACCNOIGN** is not set in your environment, DB-Access inserts two records into the **receipts** table and updates the **customer** table. Now, the decrease in the **customer** balance exceeds the sum of the inserted receipts.

But if **DBACCNOIGN** is set to `1`, messages open that indicate that DB-Access rolled back all the INSERT and UPDATE statements. The messages also identify the cause of the error so that you can resolve the problem.

LOAD statement example when DBACCNOIGN is set

You can set the **DBACCNOIGN** environment variable to protect data integrity during a LOAD statement, even if DB-Access runs the LOAD statement in menu mode.

Assume you execute the LOAD statement from the DB-Access SQL menu. Forty-nine rows of data load correctly, but the 50th row contains an invalid value that causes an error. If you set **DBACCNOIGN** to `1`, the database server does not insert the forty-nine previous rows into the database. If **DBACCNOIGN** is not set, the database server inserts the first 49 rows.

DBANSIWARN environment variable

Use the **DBANSIWARN** environment variable to indicate that you want to check for HCL Informix® extensions to ANSI-standard SQL syntax.

Unlike most environment variables, you are not required to set

```
DBANSIWARN
```

to a value. You can set it to any value or to no value.

```
setenvDBANSIWARN
```

Running DB-Access with **DBANSIWARN** set is functionally equivalent to including the **-ansi** flag when you invoke DB-Access (or any HCL Informix® product that recognizes the **-ansi** flag) from the command line. If you set **DBANSIWARN** before you run DB-Access, any syntax-extension warnings are displayed on the screen within the SQL menu.

At runtime, the **DBANSIWARN** environment variable causes the sixth character of the **sqlwarn** array in the SQL Communication Area (SQLCA) to be set to `w` when a statement is executed that is recognized as including any HCL Informix® extension to the ANSI/ISO standard for SQL syntax.

For details on SQLCA, see the *HCL® Informix® Enterprise Replication Guide*.

After you set **DBANSIWARN**, HCL Informix® extension checking is automatic until you log out or unset **DBANSIWARN**. To turn off HCL Informix® extension checking, you can disable **DBANSIWARN** with this command:

```
unsetenv DBANSIWARN
```

DBBLOBBUF environment variable

Use the **DBBLOBBUF** environment variable to control whether TEXT or BYTE values are stored temporarily in memory or in a file while being processed by the UNLOAD statement. **DBBLOBBUF** affects only the UNLOAD statement.

```
setenvDBBLOBBUFsize
```

size

represents the maximum size of TEXT or BYTE data in KB.

If the TEXT or BYTE data size is smaller than the default of 10 KB (or the setting of **DBBLOBBUF**), the TEXT or BYTE value is temporarily stored in memory. If the data size is larger than the default or the **DBBLOBBUF** setting, the data value is written to a temporary file. For instance, to set a buffer size of 15 KB, set **DBBLOBBUF** as in the following example:

```
setenv DBBLOBBUF 15
```

Here any TEXT or BYTE value smaller than 15 KB is stored temporarily in memory. Values larger than 15 KB are stored temporarily in a file.

DBCENTURY environment variable

Use the **DBCENTURY** environment variable to specify how to expand literal DATE and DATETIME values that are entered with abbreviated year values. To avoid problems in expanding abbreviated years, applications should require entry of 4-digit years, and should always display years as four digits.

```
setenvDBCENTURY{R | F|{C | P}}
```

When **DBCENTURY** is not set (or is set to `R`), the first two digits of the current year are used to expand 2-digit year values. For example, if today's date is 09/30/2003, then the abbreviated date 12/31/99 expands to 12/31/2099, and the abbreviated date 12/31/00 expands to 12/31/2000.

The R, P, F, and C settings determine algorithms for expanding two-digit years.

Setting	Algorithm
R = Current®	Use the first two digits of the current year to expand the year value.
P = Past	Expanded dates are created by prefixing the abbreviated year value with 19 and 20. Both dates are compared to the current date, and the most recent date that is earlier than the current date is used.
F = Future	Expanded dates are created by prefixing the abbreviated year value with 20 and 21. Both dates are compared to the current date, and the earliest date that is later than the current date is used.
C = Closest	Expanded dates are created by prefixing the abbreviated year value with 19, 20, and 21. These three dates are compared to the current date, and the date that is closest to the current date is used.

Settings are case sensitive, and no error is issued for invalid settings. If you enter `f` (for example), then the default (`R`) setting takes effect. The `P` and `F` settings cannot return the current date, which is not in the past or future.

Years entered as a single digit are prefixed with 0 and then expanded. Three-digit years are not expanded. Pad years earlier than 100 with leading zeros.

Related information

#unique_1975

Examples of expanding year values

The examples in this topic illustrate how various settings of **DBCENTURY** cause abbreviated years to be expanded in DATE and DATETIME values.

DBCENTURY = P

```
Example data type: DATE
Current date: 4/6/2003
User enters: 1/1/1
Prefix with "19" expansion : 1/1/1901
Prefix with "20" expansion: 1/1/2001
Analysis: Both are prior to current date, but 1/1/2001 is closer to
 current date.
```

Important: The effect of **DBCENTURY** depends on the current date from the system clock-calendar. Thus, 1/1/1, the abbreviated date in this example, would instead be expanded to 1/1/1901 if the current date were 1/1/2001 and **DBCENTURY** = P.

DBCENTURY = F

```
Example data type: DATETIME year to month
Current date: 5/7/2005
User enters: 1-1
Prefix with "20" expansion: 2001-1
Prefix with "21" expansion: 2101-1
Analysis: Only date 2101-1 is after the current date, so it is chosen.
```

DBCENTURY = C

```
Example data type: DATE
Current date: 4/6/2000
User enters: 1/1/1
Prefix with "19" expansion : 1/1/1901
Prefix with "20" expansion: 1/1/2001
Prefix with "21" expansion: 1/1/2101
Analysis: Here 1/1/2001 is closest to the current date, so it is chosen.
```

DBCENTURY = R or DBCENTURY Not Set

```
Example data type: DATETIME year to month
Current date: 4/6/2000
User enters: 1-1
Prefix with "20" expansion: 2001-1

Example data type: DATE
Current date: 4/6/2003
User enters: 0/1/1
Prefix with "20" expansion: 2000/1
Analysis: In both examples, the Prefix with "20" algorithm is used.
```

Setting **DBCENTURY** does not affect HCL Informix® products when the locale specifies a non-Gregorian calendar. The leading digits of the current year are used for alternative calendar systems when the year is abbreviated.

Abbreviated years and expressions in database objects

When an expression in a database object (including a check constraint, fragmentation expression, SPL routine, trigger, or UDR) contains a literal date or DATETIME value in which the year has one or two digits, the database server evaluates the expression using the setting that **DBCENTURY** (and other relevant environment variables) had when the database object was created (or was last modified).

If **DBCENTURY** has been reset to a new value, the new value is ignored when the abbreviated year is expanded.

For example, suppose a user creates a table and defines the following check constraint on a column named **birthdate**:

```
birthdate < '09/25/50'
```

The expression is interpreted according to the value of **DBCENTURY** when the constraint was defined. If the table that contains the **birthdate** column is created on 09/23/2000 and **DBCENTURY** `=C`, the check constraint expression is consistently interpreted as `birthdate < '09/25/1950'` when inserts or updates are performed on the **birthdate** column. Even if different values of **DBCENTURY** are set when users perform inserts or updates on the **birthdate** column, the constraint expression is interpreted according to the setting at the time when the check constraint was defined (or was last modified).

Database objects created on some earlier versions of HCL Informix® do not support the priority of creation-time settings.

For legacy objects to acquire this feature

1. Drop the objects.
2. Recreate them (or for fragmentation expressions, detach them and then reattach them).

After the objects are redefined, date literals within expressions of the objects will be interpreted according to the environment at the time when the object was created or was last modified. Otherwise, their behavior will depend on the runtime environment and might become inconsistent if this changes.

Administration of a database that includes a mix of legacy objects and new objects might become difficult because of differences between the new and the old behavior for evaluating date expressions. To avoid this, it is recommended that you redefine any legacy objects.

The value of **DBCENTURY** and the current date are not the only factors that determine how the database server interprets date and DATETIME values. The **DBDATE**, **DBTIME**, **GL_DATE**, and **GL_DATETIME** environment variables can also influence how dates are interpreted. For information about **GL_DATE** and **GL_DATETIME**, see the *HCL® Informix® GLS User's Guide*.

Important: The behavior of **DBCENTURY** for HCL Informix® is not compatible with earlier versions.

DBDATE environment variable

Use the **DBDATE** environment variable to specify the end-user formats of DATE values.

On UNIX™ systems that use the C shell, set **DBDATE** with this syntax.

```
setenvDBDATE{MD | DM | Y4 | Y2}{Y4 | Y2 | MD | DM}{/ | - | . | . | 0}
```

The following formatting symbols are valid in the **DBDATE** setting:

- . /

are characters that can exist as separators in a date format.

0

indicates that no separator is displayed between time units.

D, M

are characters that represent the day and the month.

Y2, Y4

are characters that represent the year and the precision of the year.

DBDATE can specify the following attributes of the display format:

- The order of time units (the month, day, and year) in a date
- Whether the year is shown as two digits (Y2) or four digits (Y4)
- The separator between the month, day, and year time units

For the U.S. English locale, the default for **DBDATE** is `MDY4/`, where `M` represents the month, `D` represents the day, `Y4` represents a four-digit year, and slash (**/**) is the time-units separator (for example, `01/08/2011`). Other valid characters for the separator are a hyphen (**-**), a period (**.**), or a zero (0). To indicate no separator, use the zero. The slash (**/**) is used by default if you attempt to specify a character other than a hyphen, period, or zero as a separator, or if you do not include any separator in the **DBDATE** specification.

If **DBDATE** is not set on the client, any **DBDATE** setting on the database server overrides the `MDY4/` default on the client. If **DBDATE** is set on the client, that value (rather than the setting on the database server) is used by the client.

The following table shows some examples of valid **DBDATE** settings and their corresponding displays for the date 8 January, 2011:

DBDATE Setting	Representation of January 8, 2011:		DBDATE Setting	Representation of January 8, 2011:
MDY4/	01/08/2011		Y2DM.	11.08.01
DMY2-	08-01-11		MDY20	010811
MDY4	01/08/2011		Y4MD*	2011/01/08

Formats `Y4MD`* (because asterisk is not a valid separator) and `MDY4` (with no separator defined) both display the default symbol (slash) as the separator.

Important: If you use the Y2 format, the setting of the **DBCENTURY** environment variable can also affect how literal DATE values are evaluated in data entry.

Also, certain routines that calls can use the **DBTIME** variable, rather than **DBDATE**, to set DATETIME formats to international specifications. For more information, see the discussion of the **DBTIME** environment variable in #unique_1978 and in the *HCL® Informix® Enterprise Replication Guide*.

The setting of the **DBDATE** variable takes precedence over that of the **GL_DATE** environment variable, and over any default DATE format that **CLIENT_LOCALE** specifies. For information about **GL_DATE** and **CLIENT_LOCALE**, see the *HCL® Informix® GLS User's Guide*.

End-user formats affect the following contexts:

- When you display DATE values, HCL Informix® products use the **DBDATE** environment variable to format the output.
- During data entry of DATE values, HCL Informix® products use the **DBDATE** environment variable to interpret the input.

For example, if you specify a literal DATE value in an INSERT statement, the database server expects this literal value to be compatible with the format that **DBDATE** specifies. Similarly, the database server interprets the date that you specify as the argument to the **DATE**( ) function to be in **DBDATE** format.

DATE expressions in database objects

When an expression in a database object (including a check constraint, fragmentation expression, SPL routine, trigger, or UDR) contains a literal date value, the database server evaluates the expression using the setting that **DBDATE** (or other relevant environment variables) had when the database object was created (or was last modified). If **DBDATE** has been reset to a new value, the new value is ignored when the literal DATE is evaluated.

For example, suppose **DBDATE** is set to `MDY2/` and a user creates a table with the following check constraint on the column **orderdate**:

```
orderdate < '06/25/98'
```

The date of the preceding expression is formatted according to the value of **DBDATE** when the constraint is defined. The check constraint expression is interpreted as `orderdate < '06/25/98'` regardless of the value of **DBDATE** during inserts or updates on the **orderdate** column. Suppose **DBDATE** is reset to `DMY2/` when a user inserts the value `'30/01/98'` into the **orderdate** column. The date value inserted uses the date format `DMY2/`, whereas the check constraint expression uses the date format `MDY2/`.

See Abbreviated years and expressions in database objects on page 1469 for a discussion of legacy objects from earlier versions of HCL Informix® that are always evaluated according to the runtime environment. That section describes how to redefine objects so that dates are interpreted according to environment variable settings that were in effect when the object was defined (or when the object was last modified).

Important: The behavior of **DBDATE** for HCL Informix® is not compatible with earlier versions.

DBDELIMITER environment variable

Set the **DBDELIMITER** environment variable to specify the field delimiter used with the **dbexport** utility and with the LOAD and UNLOAD statements.

```
setenvDBDELIMITER'delimiter'
```

delimiter

is the field delimiter for unloaded data files.

The *delimiter* can be any single character, except those in the following list:

- Hexadecimal digits (0 through 9,a through f, A through F)
- Newline or `CTRL-J`
- The backslash (****) symbol

The vertical bar (| `= ASCII 124`) is the default. To change the field delimiter to a plus (**+**) symbol, for example, you can set DBDELIMITER as follows:

```
setenv DBDELIMITER '+'
```

DBEDIT environment variable

Use the **DBEDIT** environment variable to specify the text editor to use with SQL statements and command files in DB-Access.

If **DBEDIT** is set, the specified text editor is invoked automatically. If **DBEDIT** is not, set you are prompted to specify a text editor as the default for the rest of the session.

```
setenvDBEDITeditor
```

editor

is the name of the text editor you want to use.

For most UNIX™ systems, the default text editor is **vi**. If you use another text editor, be sure that it creates flat ASCII files. Some word processors in *document mode* introduce printer control characters that can interfere with the operation of your HCL Informix® product.

To specify the EMACS text editor, set **DBEDIT** with the following command:

```
setenv DBEDIT emacs
```

DBFLTMASK environment variable

The utility displays the floating-point values of data types FLOAT, SMALLFLOAT, and DECIMAL(*p*) within a 14-character buffer. By default, displays as many digits to the right of the decimal point as will fit into this character buffer. Therefore, the actual number of decimal digits that displays depends on the size of the floating-point value.

To reduce the number of digits displayed to the right of the decimal point in floating-point values, set **DBFLTMASK** to the specified number of digits.

setenvDBFLTMASK*scale*

scale

is the number of decimal digits that you want the HCL Informix® client application to display in the floating-point values. Here *scale* must be smaller than 16, the default number of digits displayed.

If the floating-point value contains more digits to the right of the decimal than **DBFLTMASK** specifies, rounds the value to the specified number of digits. If the floating-point value contains fewer digits to the right of the decimal, pads the value with zeros. If you set **DBFLTMASK** to a value greater than can fit into the 14-character buffer, however, rounds the value to the number of digits that can fit.

DBINFO_DBSPACE_RETURN_NULL_FOR_INVALID_PARTNUM environment variable

Use the **DBINFO_DBSPACE_RETURN_NULL_FOR_INVALID_PARTNUM** environment variable to specify if dbinfo('dbspace', partnum) raises an error -727 or returns NULL when an invalid partition number (partnum) is provided.

```
+-DBINFO_DBSPACE_RETURN_NULL_FOR_INVALID_PARTNUM--+-'0'-+--------------------------------------+
                                                  '-'1'-'
```

A partition number is considered invalid if it resolves to a dbspace number which is not a valid dbspace in the instance. This includes the pseudo tables which are having partition numbers that would be associated with dbspace 0 which is not a (real) dbspace in an Informix instance. This reflects that pseudo tables do not directly have an on-disk representation but rather are state information from (shared) memory which are exposed via SQL.

In case of an invalid partnum the **dbinfo('dbspace', partnum)** function would result in an error '727: Invalid or NULL TBLspace number given to dbinfo(dbspace).'. When the environment variable **DBINFO_DBSPACE_RETURN_NULL_FOR_INVALID_PARTNUM** is set to 1, dbinfo() will not result in an error in this case, but rather does return NULL as dbspace name. When setting a value of '0' or not setting the environment variable, the default behavior returns an error -727 for an invalid partnum. In any case a NULL provided as partnum will result in error -727 being raised.

With SET ENVIRONMENT DBINFO_DBSPACE_RETURN_NULL_FOR_INVALID_PARTNUM the variable can be set dynamically at runtime. This overrides the current DBINFO_DBSPACE_RETURN_NULL_FOR_INVALID_PARTNUM value for the current user session only. For more information about the SET ENVIRONMENT DBINFO_DBSPACE_RETURN_NULL_FOR_INVALID_PARTNUM statement of SQL, see the Guide to SQL: Syntax.

DBLANG environment variable

Use the **DBLANG** environment variable to specify the subdirectory of **$INFORMIXDIR** or the full pathname of the directory that contains the compiled message files that HCL Informix® products use.

setenvDBLANG { *relative_path* | *full_path* }

relative_path

is a subdirectory of **$INFORMIXDIR**.

full_path

is the pathname to the compiled message files.

By default, HCL Informix® products put compiled messages in a locale-specific subdirectory of the **$INFORMIXDIR/msg** directory. These compiled message files have the file extension **.iem**. If you want to use a message directory other than **$INFORMIXDIR/msg**, where, for example, you can store message files that you create, you must perform the following steps:

To use a message directory other than $INFORMIXDIR/msg

1. Use the **mkdir** command to create the appropriate directory for the message files.

   You can make this directory under the directory **$INFORMIXDIR** or **$INFORMIXDIR/msg**, or you can make it under any other directory.
2. Set the owner and group of the new directory to **informix** and the access permission for this directory to `755`.
3. Set the **DBLANG** environment variable to the new directory. If this is a subdirectory of **$INFORMIXDIR** or **$INFORMIXDIR/msg**, then you need only list the relative path to the new directory. Otherwise, you must specify the full pathname of the directory.
4. Copy the **.iem** files or the message files that you created to the new message directory that **$DBLANG** specifies.

   All the files in the message directory should have the owner and group **informix** and access permission `644`.

HCL Informix® products that use the default U.S. English locale search for message files in the following order:

1. In **$DBLANG**, if **DBLANG** is set to a full pathname
2. In **$INFORMIXDIR/msg/$DBLANG**, if **DBLANG** is set to a relative pathname
3. In **$INFORMIXDIR/$DBLANG**, if **DBLANG** is set to a relative pathname
4. In **$INFORMIXDIR/msg/en_us/0333**
5. In **$INFORMIXDIR/msg/en_us.8859-1**
6. In **$INFORMIXDIR/msg**
7. In **$INFORMIXDIR/msg/english**

For more information about search paths for messages, see the description of **DBLANG** in the *HCL® Informix® GLS User's Guide*.

DBPATH environment variable

Use the **DBPATH** environment variable to identify the database servers that contain databases. DBPATH can also specify a list of directories (in addition to the current directory) in which looks for command scripts (**.sql** files).

The CONNECT DATABASE, START DATABASE, and DROP DATABASE statements use **DBPATH** to locate the database under two conditions:

- If the location of a database is not explicitly stated
- If the database cannot be located in the default server

The CREATE DATABASE statement does not use **DBPATH**.

To add a new **DBPATH** entry to existing entries, see Modifying an environment-variable setting on page 1443.

```
setenvDBPATH { | pathname | / /servername/full_pathname | / /servername }
```

full_pathname

is the full path, from **root**, of a directory where **.sql** files are stored.

pathname

is the valid relative path of a directory where **.sql** files are stored.

servername

is the name of a database server where databases are stored. You cannot reference database files with a *servername*.

DBPATH can contain up to 16 entries. Each entry must be less than 128 characters. In addition, the maximum length of **DBPATH** depends on the hardware platform on which you set **DBPATH**.

When you access a database with the CONNECT, DATABASE, START DATABASE, or DROP DATABASE statement, the search for the database is done first in the directory or database server specified in the statement. If no database server is specified, the default database server that was specified by the **INFORMIXSERVER** environment variable is used.

If the database is not located during the initial search, and if **DBPATH** is set, the database servers and directories in **DBPATH** are searched for in the specified database. These entries are searched in the same order in which they are listed in the **DBPATH** setting.

Using DBPATH with DB-Access

If you use and select the **Choose** option from the **SQL** menu without having already selected a database, you see a list of all the **.sql** files in the directories listed in your **DBPATH**. After you select a database, the **DBPATH** is not used to find the **.sql** files. Only the **.sql** files in the current working directory are displayed.

Searching local directories

About this task

Use a pathname without a database server name to search for **.sql** scripts on your local computer. In the following example, the **DBPATH** setting causes to search for the database files in your current directory and then in the Joachim and Sonja directories on the local computer:

```
setenv DBPATH /usr/joachim:/usr/sonja
```

As the previous example shows, if the pathname specifies a directory name but not a database server name, the directory is sought on the computer that runs the default database server that the **INFORMIXSERVER** specifies; see INFORMIXSERVER

environment variable on page 1498. For instance, with the previous example, if **INFORMIXSERVER** is set to **quality**, the **DBPATH** value is *interpreted,* as the following example shows, where the double slash precedes the database server name:

```
setenv DBPATH //quality/usr/joachim://quality/usr/sonja
```

Searching networked computers for databases

About this task

If you use more than one database server, you can set **DBPATH** explicitly to contain the database server and directory names that you want to search for databases. For example, if **INFORMIXSERVER** is set to **quality**, but you also want to search the **marketing** database server for **/usr/joachim**, set **DBPATH** as the following example shows:

```
setenv DBPATH //marketing/usr/joachim:/usr/sonja
```

Specifying a servername

About this task

You can set **DBPATH** to contain only database server names. This feature allows you to locate only databases; you cannot use it to locate command files.

The database administrator must include each database server mentioned by **DBPATH** in the **$INFORMIXDIR/etc/sqlhosts** file. For information about communication-configuration files and dbservernames, see your *HCL® Informix® Administrator's Guide* and the *HCL® Informix® Administrator's Reference*.

For example, if **INFORMIXSERVER** is set to **quality**, you can search for a database first on the **quality** database server and then on the **marketing** database server by setting **DBPATH,** as the following example shows:

```
setenv DBPATH //marketing
```

If you use in this example, the names of all the databases on the **quality** and **marketing** database servers are displayed with the **Select** option of the DATABASE menu.

DBPRINT environment variable

Use the **DBPRINT** environment variable to specify the default printing program.

setenvDBPRINT*program*

program

Any command, shell script, or UNIX™ utility that produces standard ASCII output.

If you do not set **DBPRINT**, the default *program* is found in one of two places:

- For most BSD UNIX™ systems, the default program is **lpr**.
- For UNIX™ System V, the default program is usually **lp**.

Enter the following command to set the **DBPRINT** environment variable to specify **myprint** as the print program:

```
setenv DBPRINT myprint
```

DBREMOTECMD environment variable (UNIX™)

Use the **DBREMOTECMD** environment variable to override the default remote shell to perform remote tape operations with the database server.

You can set **DBREMOTECMD** to a simple command or to a full path name.

setenvDBREMOTECMD { *command* | *pathname* }

command

A command to override the default remote shell.

pathname

A path name to override the default remote shell.

If you do not specify the full path name, the database server searches your **PATH** for the specified *command*. You should use the full path name syntax on interactive UNIX™ platforms to avoid problems with similarly named programs in other directories and possible confusion with the *restricted shell* (`/usr/bin/rsh`).

The following command sets **DBREMOTECMD** for a simple command name:

```
setenv DBREMOTECMD rcmd
```

The next command to set **DBREMOTECMD** specifies a full path name:

```
setenv DBREMOTECMD /usr/bin/remsh
```

For more information about using remote tape devices for backups, see Specify a remote device on page .

DBSPACETEMP environment variable

The **DBSPACETEMP** environment variable specifies the dbspaces in which temporary tables are built. The list can include standard dbspaces, temporary dbspaces, or both.

setenvDBSPACETEMP *dbspace*

dbspace

is the name of an existing standard or temporary dbspace.

You can list dbspaces, separated by colon (:) or comma (,) symbols, to designate space for temporary tables across physical storage devices. For example, the following command to set the **DBSPACETEMP** environment variable specifies three dbspaces for temporary tables:

```
setenv DBSPACETEMP sorttmp1:sorttmp2:sorttmp3
```

DBSPACETEMP overrides any default dbspaces that the DBSPACETEMP parameter specifies in the configuration file of the database server. For UPDATE STATISTICS operations, DBSPACETEMP is used only when you specify the HIGH keyword option.

On UNIX™ platforms, you might have better performance if the list of dbspaces in **DBSPACETEMP** is composed of chunks that are allocated as raw devices.

The number of dbspaces is limited by the maximum size of the environment variable, as defined by your operating system. Your database server does not create a dbspace specified by the environment variable if the dbspace does not exist.

The two classes of temporary tables are *explicit* temporary tables that the user creates and *implicit* temporary tables that the database server creates. Use **DBSPACETEMP** to specify the dbspaces for both types of temporary tables.

If you create an explicit temporary table with the CREATE TEMP TABLE statement and do not specify a dbspace for the table either in the IN *dbspace* clause or in the FRAGMENT BY clause, the database server uses the settings in **DBSPACETEMP** to determine where to create the table.

If you create an explicit temporary table with the SELECT INTO TEMP statement, the database server uses the settings in **DBSPACETEMP** to determine where to create the table.

If **DBSPACETEMP** is set, and the dbspaces that it lists include both logging and non-logging dbspaces, the database server stores temporary tables that implicitly or explicitly support transaction logging in a logged dbspace, and non-logging temporary tables in a non-logging dbspace.

The database server creates implicit temporary tables for its own use while executing join operations, SELECT statements with the GROUP BY clause, SELECT statements with the ORDER BY clause, and index builds.

When it creates explicit or implicit temporary tables, the database server uses disk space for writing the temporary data. If there are conflicts among settings or statement specifications for the location of a temporary table, these conflicts are resolved in this descending (highest to lowest) order of precedence:

1. On UNIX™ platforms, the operating-system directory or directories that the environment variable **PSORT_DBTEMP** specifies, if this is set
2. The dbspace or dbspaces that the environment variable **DBSPACETEMP** specifies, if this is set
3. The dbspace or dbspaces that the ONCONFIG parameter DBSPACETEMP specifies.
4. The operating-system file space specified by the DUMPDIR configuration parameter
5. The directory $INFORMIXDIR/tmp (UNIX™) or $INFORMIXDIR\tmp (Windows™).

Important: If the **DBSPACETEMP** environment variable is set to an invalid value, the database server defaults to the root dbspace for explicit temporary tables and to **/tmp** for implicit temporary tables, rather than to the setting of the DBSPACETEMP configuration parameter. In this situation, the database server might fill **/tmp** to the limit and eventually bring down the database server or kill the file system.

DBTEMP environment variable

The **DBTEMP** environment variable is used by DB-Access. **DBTEMP** resembles **DBSPACETEMP**, specifying the directory in which to place temporary files and temporary tables.

```
setenv DBTEMP pathname
```

pathname

The full path name of the directory for temporary files and tables.

For DB-Access to work correctly on Windows™ platforms, **DBTEMP** should be set to `$INFORMIXDIR/infxtmp`.

The following example sets **DBTEMP** to the path name `usr/magda/mytemp` for UNIX™ systems that use the C shell:

```
setenv DBTEMP usr/magda/mytemp
```

Important: DBTEMP can point to an NFS-mounted directory only if the vendor of that NFS device is certified by IBM®.

If **DBTEMP** is not set, the database server creates temporary files in the `/tmp` directory and temporary tables in the **DBSPACETEMP** directory. See DBSPACETEMP environment variable on page 1477 for the default if **DBSPACETEMP** is not set. Similarly, if you do not set **DBTEMP** on the client system, temporary files (such as those created for scroll cursors) are created in the `/tmp` directory.

You might experience unexpected behavior or failure in operations on values of large or complex data types, such as BYTE or ROW, if DBTEMP is not set.

DBUPSPACE environment variable

Use the **DBUPSPACE** environment variable to specify the amount of system disk space and the amount of memory that the UPDATE STATISTICS MEDIUM and UPDATE STATISTICS HIGH statement can use when it reads and sorts column values to construct column distributions. The **DBUPSPACE** setting can also request SET EXPLAIN output to describe the execution path for calculating the statistical distributions.

setenvDBUPSPACE { **1024** | [*disk*] } { : **15** | [: *memory*] } [: *directive*]

disk

is an unsigned integer, specifying the disk space (in KiB) to allocate for sorting in UPDATE STATISTICS MEDIUM and HIGH operations.

memory

is an unsigned integer, specifying the maximum amount of sorting memory (in MiB, in the range from 4 to 50 megabytes) to allocate without using PDQ.

directive

is an unsigned integer, encoding one of the following directives for the UPDATE STATISTICS execution plan:

- 1: Do not use any indexes for sorting. Print the entire plan for update statistics in the `sqexplain.out` file.
- 2: Do not use any indexes for sorting. Do not print the plan for update statistics.
- 3 or greater: Use available indexes for sorting. Print the entire plan for update statistics in explain output file.

For example, to set **DBUPSPACE** to 2,500 KiB of disk space and 1 megabyte of memory, enter this command:

```
setenv DBUPSPACE 2500:1
```

After you set this value, the database server will attempt to use no more than 2,500 KiB of disk space during the execution of an UPDATE STATISTICS MEDIUM or HIGH statement. If a table requires 5 megabytes of disk space for sorting, then UPDATE STATISTICS accomplishes the task in two passes; the distributions for one half of the columns are constructed with each pass. For a table of a given storage size, this parameter determines the number of passes, but no pass can write less than a full column.

If you do not set **DBUPSPACE**, the default setting is 1 megabyte (1,024 KiB) for *disk*, and 15 megabytes for *memory*. If you attempt to set the first **DBUPSPACE** parameter to any value less than 1,024 KiB, it is automatically set to 1,024 KiB, but no error message is returned. If this *disk* value is not large enough to allow more than one distribution to be constructed at a time, at least one distribution is done, even if the amount of disk space required to do this is more than what **DBUPSPACE** specifies. That is, regardless of the *disk* parameter setting for **DBUPSPACE**, the largest individual column storage requirement of a table determines the actual upper limit on disk space for a single pass in any UPDATE STATISTICS HIGH or MEDIUM operation.

Related information

Default name and location of the explain output file on UNIX on page

Default name and location of the output file on Windows on page

DEFAULT_ATTACH environment variable

The **DEFAULT_ATTACH** environment variable supports the legacy behavior of Version 7.x of HCL Informix®, in which the pages of nonfragmented B-tree indexes on nonfragmented tables were stored, by default, in the same dbspace partition as the data pages. (The name "**DEFAULT_ATTACH**" derives from an obsolete definition of an *attached index*, a term that now refers to an index whose fragmentation strategy is the same as the fragmentation strategy of its table. Do not confuse the obsolete Version 7.x definition with this current definition.)

```
setenvDEFAULT_ATTACH1
```

If the **DEFAULT_ATTACH** environment variable is set to 1, then by default, the pages of nonfragmented B-tree indexes on nonfragmented tables are stored in the same partition (and in the same dbspace) that stores data pages of the table. The IN TABLE keywords of the CREATE INDEX statement are not required (but do not return an error).

Setting **DEFAULT_ATTACH** to 1 has no effect, however, on any other types of indexes, whose pages are always stored in separate partitions from the data pages of the indexed table. These index types whose storage distribution is always different from that of their table include

- R-tree indexes,
- functional indexes,
- forest of trees indexes,
- fragmented indexes,
- and indexes on fragmented tables.

Index storage in the same partition as the data pages is supported only for nonfragmented B-tree indexes on nonfragmented tables.

If **DEFAULT_ATTACH** is not set, then by default, any CREATE INDEX statement that does not specify IN TABLE as its Storage Options clause creates an index whose pages are stored in partitions separate from the data pages. This release of HCL Informix® can support existing indexes that were created by Version 7.x of HCL Informix®.

Important: Future releases of HCL Informix® might not continue to support **DEFAULT_ATTACH.** Developing new applications that depend on this deprecated feature is not recommended.

DELIMIDENT environment variable

The **DELIMIDENT** environment variable specifies that strings enclosed between double quotation (") marks are delimited database identifiers.

The **DELIMIDENT** environment variable is also supported on client systems, where it can be set to y, to n, or to no setting.

- **y** specifies that client applications must use single quotation (') symbols to delimit character strings, and must use double quotation (") symbols only around delimited SQL identifiers, which can support a larger character set than is valid in undelimited identifiers. Letters within delimited strings or delimited identifiers are case-sensitive. This is the default value for OLE DB and .NET.
- **n** specifies that client applications can use double quotation (") or single quotation (') symbols to delimit character strings, but not to delimit SQL identifiers. If the database server encounters a string delimited by double or single quotation symbols in a context where an SQL identifier is required, it issues an error. An owner name that qualifies an SQL identifier can be delimited by single quotation (') symbols. You must use a pair of the same quotation symbols to delimit a character string.

  This is the default value for ESQL/C, JDBC, and ODBC. APIs that have ESQL/C as an underlying layer, such as the (LIBDMI), and the C++ API, behave as ESQL/C, and use 'n' as the default if no value for DELIMIDENT is specified on the client system.
- Specifying the DELIMIDENT environment variable with no value on the client system requires client applications to use the DELIMIDENT setting that is the default for their application programming interface (API).

```
setenvDELIMIDENT
```

No value is required; **DELIMIDENT** takes effect if it exists, and it remains in effect while it is on the list of environment variables. Removing DELIMIDENT when it is set at the server level requires restarting the server.

Delimited identifiers can include white space (such as the phrase "**Vitamin E**") or can be identical to SQL keywords, (such as "**TABLE**" or "**USAGE**"). You can also use them to declare database identifiers that contain characters outside the default character set for SQL identifiers (such as "**Column #6**"). In the default locale, this set consists of letters, digits, and the underscore (_) symbol.

Even if DELIMIDENT is set, you can use single quotation (') symbols to delimit authorization identifiers as the owner name component of a database object name, as in the following example:

```
RENAME COLUMN 'Owner'.table2.collum3 TO column3;
```

This example is an exception to the general rule that when **DELIMIDENT** is set, the SQL parser interprets character strings delimited by single quotation symbols as string literals, and interprets character strings delimited by double quotation symbols (") as SQL identifiers.

Database identifiers (also called *SQL identifiers*) are names for database objects, such as tables and columns. *Storage identifiers* are names for storage objects, such as dbspaces, blobspaces, and sbspaces. You cannot use **DELIMIDENT** to declare storage identifiers that contain characters outside the default SQL character set.

Delimited identifiers are case sensitive. To use delimited identifiers, applications in must set **DELIMIDENT** at compile time and at run time.

Important: If **DELIMIDENT** is not already set, you should be aware that setting it can cause the failure of existing `.sql` scripts or client applications that use double (") quotation marks in contexts other than delimiting SQL identifiers, such as delimiters of string literals. You must use single (') rather than double quotation marks for delimited constructs that are not SQL identifiers if **DELIMIDENT** is set.

On UNIX™ systems that use the C shell and on which **DELIMIDENT** has been set, you can disable this feature (which causes anything between double quotation symbols to be interpreted as an SQL identifier) by the command:

```
unsetenv  DELIMIDENT
```

ENVIGNORE environment variable (UNIX™)

The **ENVIGNORE** environment variable can deactivate specified environment variable settings in the common (shared) configuration file, `informix.rc`, and private environment-configuration file, `.informix`.

setenvENVIGNORE *variable*

variable

The name of an environment variable to be deactivated.

Use colon (**:**) symbols between consecutive *variable* names. For example, to ignore the **DBPATH** and **DBMONEY** entries in the environment-configuration files, enter the following command:

```
setenv ENVIGNORE DBPATH:DBMONEY
```

The common environment-configuration file is stored in `$INFORMIXDIR/etc/informix.rc`.

The private environment-configuration file is stored in the home directory of the user as `.informix`.

For information about creating or modifying an environment-configuration file, see Setting environment variables in a configuration file on page 1442.

ENVIGNORE itself cannot be set in an environment-configuration file.

FET_BUF_SIZE environment variable

The **FET_BUF_SIZE** environment variable can override the default setting for the size of the fetch buffer for all data types except BYTE and TEXT values. For ANSI databases, you must set transactions to READ ONLY for the **FET_BUF_SIZE**environment variable to improve performance, otherwise rows are returned one by one.

```
setenvFET_BUF_SIZEsize
```

size

is a positive integer that is larger than the default buffer size, but no greater than 2147483648 (2GB), specifying the size (in bytes) of the fetch buffer that holds data retrieved by a query.

For example, to set a buffer size to `5,000` bytes on a UNIX™ system that uses the C shell, set **FET_BUF_SIZE** by entering the following command:

```
setenv FET_BUF_SIZE 5000
```

When **FET_BUF_SIZE** is set to a valid value, the new value overrides the default value (or any previously set value of **FET_BUF_SIZE**). The default setting for the fetch buffer is dependent on row size.

The processing of BYTE and TEXT values is not affected by **FET_BUF_SIZE**.

No error is raised if **FET_BUF_SIZE** is set to a value that is less than the default size or is larger than 2147483648 (2GB). In these cases, however, the invalid fetch buffer size is ignored, and the default size is in effect.

A valid **FET_BUF_SIZE** setting is in effect for the local database server and for any remote database server from which you retrieve rows through a distributed query in which the local server is the coordinator and the remote database is subordinate. The greater the size of the buffer, the more rows can be returned, and the less frequently the client application must wait while the database server returns rows. A large buffer can improve performance by reducing the overhead of filling the client-side buffer.

IFMXMONGOAUTH environment variable

Set the **IFMXMONGOAUTH** environment variable to enable PAM authentication for MongoDB clients through the wire listener.

You can set the **IFMXMONGOAUTH** environment variable to any value or to no value.

```
setenv IFMXMONGOAUTH 1
```

Setting the **IFMXMONGOAUTH** environment variable is a prerequisite to configuring PAM authentication for MongoDB clients.

You can disable the **IFMXMONGOAUTH** environment variable with this command:

```
unsetenv IFMXMONGOAUTH
```

Related information

Configuring PAM authentication on page

IFX_DEF_TABLE_LOCKMODE environment variable

The **IFX_DEF_TABLE_LOCKMODE** environment variable can specify the default lock mode for database tables that are subsequently created without explicitly specifying the LOCKMODE PAGE or LOCKMODE ROW keywords. This feature is convenient if you must create several tables of the same lock mode. UNIX™ systems that use the C shell support the following syntax:

```
setenvIFX_DEF_TABLE_LOCKMODE { PAGE  | ROW }
```

PAGE

The default lock mode is page-level granularity. This value disables the LAST COMMITTED feature of COMMITTED READ.

ROW

The default lock mode is row-level granularity.

Similar functionality is available by setting the DEF_TABLE_LOCKMODE parameter of the ONCONFIG file to PAGE or ROW. When a table is created or modified, any conflicting lock mode specifications are resolved according to the following descending (highest to lowest) order of precedence:

1. Explicit LOCKMODE specification of CREATE TABLE or ALTER TABLE
2. **IFX_DEF_TABLE_LOCKMODE** environment variable setting
3. DEF_TABLE_LOCKMODE parameter setting in the ONCONFIG file
4. The system default lock mode (= page mode)

To make the DEF_TABLE_LOCKMODE setting the default mode (or to restore the system default if DEF_TABLE_LOCKMODE is not set) use the command:

```
unsetenv IFX_DEF_TABLE_LOCKMODE
```

If **IFX_DEF_TABLE_LOCKMODE** is set in the environment of the database server before running **oninit**, then its scope is all sessions of the database server (just as if DEF_TABLE_LOCKMODE were set in the ONCONFIG file). If **IFX_DEF_TABLE_LOCKMODE** is set in the shell, or in the **$HOME/.informix** or **$INFORMIXDIR/etc/informix.rc** files, then the scope is restricted to the current session (if you set it in the shell) or to the individual user.

Important: This has no effect on existing tables. If you specify *ROW* as the lock mode, the database will use this to restore, recover, or copy data. For tables that were created in *PAGE* mode, this might cause lock-table overflow or performance degradation.

IFX_DIRECTIVES environment variable

The **IFX_DIRECTIVES** environment variable setting determines whether the optimizer allows query optimization directives from within a query. The **IFX_DIRECTIVES** environment variable is set on the client.

You can specify either `ON` and `OFF` or `1` and `0` to set the environment variable.

```
setenvIFX_DIRECTIVES{1 | 0}
```

1

Optimizer directives accepted

0

Optimizer directives not accepted

The setting of the **IFX_DIRECTIVES** environment variable overrides the value of the DIRECTIVES configuration parameter that is set for the database server. If the **IFX_DIRECTIVES** environment variable is not set, however, then all client sessions will inherit the database server configuration for directives that the ONCONFIG parameter DIRECTIVES determines. The default setting for the **IFX_DIRECTIVES** environment variable is `ON`.

For more information about the DIRECTIVES parameter, see the *HCL® Informix® Administrator's Reference*. For more information about the performance impact of directives, see your .

IFX_EXTDIRECTIVES environment variable

The **IFX_EXTDIRECTIVES** environment variable specifies whether the query optimizer allows external query optimization directives from the **sysdirectives** system catalog table to be applied to queries in existing applications.

You have two options for setting the **IFX_EXTDIRECTIVES** environment variable:

- Global, for all users:

  On the server, set **IFX_EXTDIRECTIVES** in the environment as user informix and then run the oninit command.

- Client specific:

  On the client, set **IFX_EXTDIRECTIVES** in the environment. When **IFX_EXTDIRECTIVES** is set in the client environment, the client setting are used regardless of the server (global) setting.

You can determine the server setting using the onstat -g env command.

You can specify either `ON` and `OFF` or `1` and `0` to set the environment variable.

```
setenvIFX_DIRECTIVES{1 | 0}
```

1

External optimizer directives accepted

0

External optimizer directives not accepted

Queries within a given client application can use external directives if both the EXT_DIRECTIVES parameter in the configuration file of the database server and the **IFX_EXTDIRECTIVES** environment variable setting on the client system are both set to 1 or ON. If **IFX_EXTDIRECTIVES** is not set, external directives are supported only if the ONCONFIG parameter EXT_DIRECTIVES is set to 2. The following table summarizes the effect of valid IFX_EXTDIRECTIVES and EXT_DIRECTIVES settings on support for external optimizer directives.

Table 83. Effect of IFX_EXTDIRECTIVES and EXT_DIRECTIVES settings on external directives

	EXT_DIRECTIVES = 0	EXT_DIRECTIVES = 1	EXT_DIRECTIVES = 2
IFX_EXTDIRECTIVES No setting	OFF	OFF	ON
IFX_EXTDIRECTIVES0 = OFF	OFF	OFF	OFF
IFX_EXTDIRECTIVES1 = ON	OFF	ON	ON

The database server interprets any EXT_DIRECTIVES setting besides 1 or 2 (or no setting) as equivalent to OFF, disabling support for external directives. Any value of **IFX_EXTDIRECTIVES** other than 1 has the same effect for the client.

For information about how to define external optimizer directives, see the description of the SAVE EXTERNAL DIRECTIVES statement of SQL in the *HCL® Informix® Guide to SQL: Syntax*. For more information about the EXT_DIRECTIVES configuration parameter, see the *HCL® Informix® Administrator's Reference*. For more information about the performance impact of directives, see your .

IFX_LARGE_PAGES environment variable

The **IFX_LARGE_PAGES** environment variable specifies whether the database server can use large pages on platforms where the hardware and the operating system support large pages of shared memory. If this is enabled in the server environment, HCL Informix® can use the large pages for non-message shared memory segments that are located in physical memory.

The **IFX_LARGE_PAGES** environment variable is supported only on AIX®, Solaris, and Linux™ operating systems. The setting of **IFX_LARGE_PAGES** has no effect on Informix® if the operating system does not support large pages, or if large pages are not configured on the system.

You can specify either `1` or `0` to set this environment variable.

```
setenvIFX_LARGE_PAGES{1 | 0}
```

0

The use of large pages is disabled. This is the default on AIX® systems.

1

The use of large pages is enabled. This is the default on Solaris and Linux™ systems.

The DBSA must use operating system commands to configure the large pages. See the operating system documentation for the configuration procedures.

Informix® can use large pages for non-message shared memory segments that are locked in physical memory, if sufficient large pages are configured and available. The RESIDENT configuration parameter controls whether a shared memory segment is locked in physical memory, so that the segment cannot be swapped. If there are insufficient large pages to hold a segment, the segment might contain a mixture of large pages and regular pages.

On AIX® the large pages used by Informix® are 16 MB in size.

On Linux™ x86_64 the large pages used by Informix® are defined by the `Hugepagesize` entry in the `/proc/meminfo` file.

Informix® aligns the segment address and rounds up to the segment size automatically. In addition to messages regarding rounding, the server prints an informational message to the server log file whenever it attempts to use large pages to store a segment.

When **IFX_LARGE_PAGES** is enabled, the use of large pages can offer significant performance benefits in large memory configurations.

Related information

RESIDENT configuration parameter on page

IFX_LOB_XFERSIZE environment variable

Use the **IFX_LOB_XFERSIZE** environment variable to specify the number of bytes in a CLOB or BLOB data type to transfer from a client application to the database server before checking whether an error has occurred.

The error check occurs each time the specified number of bytes is transferred. If an error occurs, the remaining data is not sent and an error is reported. If no error occurs, the file transfer will continue until it finishes.

For example, if the value of **IFX_LOB_XFERSIZE** is set to 10485760 (10 MB), then error checking will occur after every 10485760 bytes of the CLOB or BLOB data is sent. If **IFX_LOB_XFERSIZE** is not set, the error check occurs after the entire BLOB or CLOB data is transferred.

The valid range for **IFX_LOB_XFERSIZE** is from 1 to 9223372036854775808 bytes. The **IFX_LOB_XFERSIZE** environment variable is set on the client.

```
setenvIFX_LOB_XFERSIZEvalue
```

value

the number of bytes in a CLOB or BLOB to transfer from a client application to the database server before checking whether an error has occurred

You should adjust the value of **IFX_LOB_XFERSIZE** to suit your environment. Set **IFX_LOB_XFERSIZE** low enough so that transmission errors of large BLOB or CLOB data types are detected early, but not so low that excessive network resources are used.

IFX_LONGID environment variable

The **IFX_LONGID** environment variable setting and the version number of the client application determine whether a given client application is capable of handling long identifiers. (Older versions of HCL Informix® restricted SQL identifiers to 18 or fewer bytes; *long identifiers* can have up to 128 bytes when **IFX_LONGID** is set.) Valid **IFX_LONGID** values are `1` and `0`.

```
setenvIFX_LONGID{1 | 0}
```

1

Client supports long identifiers.

0

Client cannot support long identifiers.

When **IFX_LONGID** is set to zero, applications display only the first 18 bytes of long identifiers, without indicating (by `+`) that truncation has occurred.

If **IFX_LONGID** is unset or is set to a value other than `1` or `0`, the determination is based on the internal version of the client application. If the (server-based) version is not less than 9.0304, or is in the (CSDK-based) range 2.90 ≤ *version* < 4.0, the client is considered capable of handling long identifiers. Otherwise, the client application is considered incapable.

The **IFX_LONGID** setting overrides the internal version of the client application. If the client cannot handle long identifiers despite a newer version number, set **IFX_LONGID** to `0`. If the client version can handle long identifiers despite an older version number, set **IFX_LONGID** to `1`.

If you set **IFX_LONGID** on the client, the setting affects only that client. If you start the database server with **IFX_LONGID** set, all client applications use that setting by default. If **IFX_LONGID** is set to different values on the client and on the database server, however, the client setting takes precedence.

Important: ESQL executables that have been built with the -static option using the `libos.a` library version that does not support long identifiers cannot use the **IFX_LONGID** environment variable. You must recompile such applications with the new `libos.a` library that includes support for long identifiers. Executables that use shared libraries (no -static option) can use **IFX_LONGID** without recompilation provided that they use the new `libifos.so` that provides support for long identifiers. For details, see your ESQL product publication.

IFX_NETBUF_PVTPOOL_SIZE environment variable (UNIX™)

Use the **IFX_NETBUF_PVTPOOL_SIZE** environment variable to specify the maximum size of the free (unused) private network buffer pool for each database server session.

```
setenvIFX_NETBUF_PVTPOOL_SIZEcount
```

count

an integer specifying the number of units (buffers) in the pool.

The default size is 1 buffer. If **IFX_NETBUF_PVTPOOL_SIZE** is set to 0, then each session obtains buffers from the free global network buffer pool. You must specify the value in decimal form.

IFX_NETBUF_SIZE environment variable

Use the **IFX_NETBUF_SIZE** environment variable to configure the network buffers to the optimum size. This environment variable specifies the size of all network buffers in the free (unused) global pool and the private network buffer pool for each database server session.

```
setenvIFX_NETBUF_SIZEsize
```

size

is the integer size (in bytes) for one network buffer.

The default size is 4 KB (4,096 bytes). The maximum size is 64 KB (65,536 bytes) and the minimum size is 512 bytes. You can specify the value in hexadecimal or decimal form.

Tip: You cannot set a different size for each session.

IFX_NO_SECURITY_CHECK environment variable (UNIX™)

The **IFX_NO_SECURITY_CHECK** environment variable allows user **informix** or **root** to complete operations with a database server instance even when the HCL Informix® utilities detect that the `$INFORMIXDIR` path is not secure. Do not use this environment variable unless your system setup makes it absolutely necessary to do so.

The purpose of **IFX_NO_SECURITY_CHECK** is for environments where the database server started but while running it detects that the runtime path is not secure anymore. In this case, a superuser might be required to stop the database server in order to remedy the security flaw. With this environment variable, either user **informix** or **root** can use the onmode utility to shut down a nonsecure Informix® instance, which would otherwise not be possible because key programs do not run when the `$INFORMIXDIR` path is not secure.

There is some risk in using this environment variable, but in some circumstances it might be necessary to remedy a bigger security problem. The requirement that only user **informix** or **root** can invoke **IFX_NO_SECURITY_CHECK** makes it unlikely that an illegitimate user would be able to run it.

To use this environment variable, set it to any non-empty string.

```
setenvIFX_NO_SECURITY CHECK1
```

1

Any value entered here when running this environment variable disables the onsecurity utility.

Important: Turn off this environment variable after you have finished troubleshooting the security problem.

IFX_NO_TIMELIMIT_WARNING environment variable

Trial or evaluation versions of HCL Informix® software products, which cease to function when some time limit has elapsed since the software was installed, by default issue warning messages that tell users when the license will expire. If you set the **IFX_NO_TIMELIMIT_WARNING** environment variable, however, the time-limited software does not issue these warning messages.

```
setenvIFX_NO_TIMELIMIT_WARNING
```

For users who dislike viewing warning messages, this feature is an alternative to redirecting the error output. Setting **IFX_NO_TIMELIMIT_WARNING** has no effect, however, on when a time-limited license expires; the software ceases to function at the same point in time when it would if this environment variable had not been set. If you do set **IFX_NO_TIMELIMIT_WARNING**, users will not see potentially annoying warnings about the impending license expiration, but some users might be annoyed at you when the database server (or whatever software has a time-limited license) ceases to function without any warning.

IFX_NODBPROC environment variable

The **IFX_NODBPROC** environment variable lets you prevent the database server from running the sysdbopen( ) or sysdbclose( ) procedure. These procedures cannot be run if this environment variable is set to any value.

```
setenvIFX_NODBPROCstring
```

string

Any value prevents the database server from running sysdbopen( ) or sysdblcose( ).

IFX_NOT_STRICT_THOUS_SEP environment variable

HCL Informix® requires the thousands separator to have 3 digits following it. For example, 1,000 is considered correct, and 1,00 is considered wrong. In previous releases, both formats were considered correct.

```
setenvIFX_NOT_STRICT_THOUS_SEPn
```

n

Set *n* to 1 for the behavior in previous releases, which is that the thousands separator can have fewer than three digits following it.

IFX_ONTAPE_FILE_PREFIX environment variable

When TAPEDEV and LTAPEDEV specify directories, use the **IFX_ONTAPE_FILE_PREFIX** environment variable to specify a prefix for backup file names that replaces the *hostname_servernum* format. If no value is set, file names are *hostname_servernum*_L*n* for levels and *hostname_servernum*_Log*nnnnnnnnn* for log files.

If you set the value of IFX_ONTAPE_FILE_PREFIX to My_Backup, the backup file names have the following names:

- My_Backup_L0
- My_Backup_L1
- My_Backup_L2
- My_Backup_Log0000000001
- My_Backup_Log0000000002

```
setenvIFX_ONTAPE_FILE_PREFIXstring
```

string

The prefix to use for the names of backup files.

IFX_PAD_VARCHAR environment variable

The **IFX_PAD_VARCHAR** environment variable setting controls how the database server sends and receives VARCHAR and NVARCHAR data values. Valid **IFX_PAD_VARCHAR** values are `1` and `0`.

```
setenvIFX_PAD_VARCHAR { 1  | 0 }
```

1

Transmit the entire structure, up to the declared *max* size.

0

Transmit only the portion of the structure containing data.

For example, to send the string "ABC" from a column declared as NVARCHAR(255) when **IFX_PAD_VARCHAR** is set to 0 would send 3 bytes.

If the setting were 1 in the previous example, however, the number of bytes sent would be 255 bytes.

The effect **IFX_PAD_VARCHAR** is context-sensitive. In a low-bandwidth network, a setting of 0 might improve performance by reducing the total volume of transmitted data. But in a high-bandwidth network, a setting of 1 might improve performance, if the CPU time required to process variable-length packets were greater than the time required to send the entire character stream. In cross-server distributed operations, this setting has no effect, and padding characters are dropped from VARCHAR or NVARCHAR values that are passed between database servers.

IFX_UPDDESC environment variable

You must set the **IFX_UPDDESC** environment variable at execution time before you can do a DESCRIBE of an UPDATE statement.

```
setenvIFX_UPDDESCvalue
```

value

is any non-NULL value.

A NULL value (here meaning that **IFX_UPDDESC** is not set) disables the describe-for-update feature. Any non-NULL value enables the feature.

IFX_VALIDATE_STORAGEPOOLPATH environment variable

The **IFX_VALIDATE_STORAGEPOOLPATH** environment variable provides customer a choice to skip the storage pool path validation.

It has the following syntax.

```
setenv IFX_VALIDATE_STORAGEPOOLPATH { 1  | 0 }
```

value

1 (or enable) to validate storage pool path. This is the default value.

value

0 (or disable) to disable storage pool path validation

To avoid the storage pool creation at /dev which is taken as a default path when the raw devices are used, we use this setting. So, when IFX_VALIDATE_STORAGEPOOLPATH=1, storage pool path will be checked. When IFX_VALIDATE_STORAGEPOOLPATH=0 , customer gets a choice to ignore storage pool path validation, thereby creating the storage pool in /dev. By default, we will always validate the path.

IFX_XASTDCOMPLIANCE_XAEND environment variable

In earlier releases of HCL Informix®, an internal rollback of a global transaction freed the transaction. In releases later than Version 9.40, however, the default behavior after an internal rollback is not to free the global transaction until an explicit rollback, as required by the X/Open XA standard. By setting the DISABLE_B162428_XA_FIX configuration parameter to 1, you can restore the legacy behavior as the default for all sessions.

The **IFX_XASTDCOMPLIANCE_XAEND** environment variable can override the configuration parameter for the current session, using the following syntax. Valid **IFX_XASTDCOMPLIANCE_XAEND** values are `1` and `0`.

```
setenvIFX_XASTDCOMPLIANCE_XAEND { 1  | 0 }
```

0

Frees global transactions only after an explicit rollback

1

Frees global transactions after any rollback

This environment variable can be particularly useful when the server instance is disabled for new behavior by the DISABLE_B162428_XA_FIX configuration parameter, but one client requires the new behavior. Setting this environment variable to zero supports the new behavior in the current session.

IFX_XFER_SHMBASE environment variable

An alternative base address for a utility to attach the server shared memory segments.

```
setenvIFX_XFER_SHMBASE { address }
```

address

Valid address in hexadecimal

After the database server allocates shared memory, the database server might allocate multiple contiguous OS shared memory segments. The client utility that connects to shared memory must attach all those OS segments contiguously also. The utility might have some other shared objects (for example, the xbsa library in onbar) loaded at the address where the server has shared memory segment attached. To workaround this situation, you can specify a different base address in the environment variable IFX_XFER_SHMBASE for the utility to attach the shared memory segments. The onstat, onmode, and oncheck utilities must attach to exact same shared memory base as oninit. Setting IFX_XFER_SHMBASE is not an option for these utilities.

INFORMIXC environment variable (UNIX™)

The **INFORMIXC** environment variable specifies the filename or pathname of the C compiler to be used to compile files that generates. The setting takes effect only during the C compilation stage.

If **INFORMIXC** is not set, the default compiler on most systems is **cc**.



Tip: On Windows™, you pass either -mcc or -bcc options to the *esql* preprocessor to use either the Microsoft™ or Borland C compilers.

setenvINFORMIXC { *compiler* | *pathname* }

compiler

The file name of the C compiler.

pathname

The full path name of the C compiler.

For example, to specify the GNU C compiler, enter the following command:

```
setenv INFORMIXC gcc
```



Important: If you use **gcc**, be aware that the database server assumes that strings are writable, so you must compile by using the -fwritable-strings option. Failure to do so can produce unpredictable results, possibly including core dumps.

INFORMIXCMNAME environment variable

If the Connection Manager raises an event alarm, the INFORMIXCMNAME environment variable is used to store the name of the Connection Manager instance that raised the alarm. The environment variable is set automatically by the Connection Manager.

The INFORMIXCMNAME environment variable corresponds to the NAME parameter in the Connection Manager configuration file. The environment variable is used by the CMALARMPROGRAM program to determine the Connection Manager instance

responsible for the event alarm. You can also use the environment variable in your own Connection Manager event alarm handler.

The environment variable is set automatically by the Connection Manager and should not be modified.

Related information

INFORMIXCMCONUNITNAME environment variable on page 1494

The oncmsm utility on page

Connection Manager event alarm IDs on page

INFORMIXCMCONUNITNAME environment variable

If the Connection Manager raises an event alarm, the INFORMIXCMCONUNITNAME environment variable is used to store the name of the Connection Manager connection unit that raised the alarm. The environment variable is set automatically by the Connection Manager.

The INFORMIXCMCONUNITNAME environment variable corresponds to the connection unit name parameter in the Connection Manager configuration file. The environment variable is used by the CMALARMPROGRAM program to determine the Connection Manager instance responsible for the event alarm. You can also use the environment variable in your own Connection Manager event alarm handler.

The environment variable is set automatically by the Connection Manager and should not be modified.

Related information

INFORMIXCMNAME environment variable on page 1493

The oncmsm utility on page

Connection Manager event alarm IDs on page

INFORMIXCONCSMCFG environment variable

Use the **INFORMIXCONCSMCFG** environment variable to specify the location of the **concsm.cfg** file that describes communications support modules.

setenvINFORMIXCONCSMCFG*pathname*

pathname

specifies the full pathname of the **concsm.cfg** file.

The following command specifies that the **concsm.cfg** file is in **/usr/myfiles**:

```
setenv INFORMIXCONCSMCFG /usr/myfiles
```

You can also specify a different name for the file. The following example specifies a filename of **csmconfig** in the same directory:

```
setenv INFORMIXCONCSMCFG /usr/myfiles/csmconfig
```

The default location of the **concsm.cfg** file is in **$INFORMIXDIR/etc**. For more information about communications support modules and the contents of the **concsm.cfg** file, see the *HCL® Informix® Administrator's Reference*.

INFORMIXCONRETRY environment variable

The **INFORMIXCONRETRY** environment variable sets a limit on the maximum number of connection attempts that can be made to each database server by the client after the initial connection attempt fails. These attempts are made within the time limit that the **INFORMIXCONTIME** setting specifies.

setenvINFORMIXCONRETRY*count*

count

The number of additional attempts to connect to each database server after the initial connection attempt fails.

For example, the following command sets **INFORMIXCONRETRY** to specify three connection attempts after the initial attempt:

```
setenv INFORMIXCONRETRY 3
```

The default value for **INFORMIXCONRETRY** is one attempt after the initial connection attempt.

Order of precedence among INFORMIXCONRETRY settings

When you specify a setting for the **INFORMIXCONRETRY** client environment variable, it overrides any **INFORMIXCONRETRY** configuration parameter setting in the `onconfig` file.

If the SET ENVIRONMENT statement specifies a setting for the INFORMIXCONRETRY session environment option, however, the SQL statement setting overrides the **INFORMIXCONRETRY** client environment variable setting for subsequent connection attempts during the current session. The SET ENVIRONMENT INFORMIXCONRETRY setting has no effect on other sessions.

In summary, this is the ascending order (lowest to highest) of the methods for setting a limit on attempts for a connection to a database server:

- **INFORMIXCONRETRY** configuration parameter
- **INFORMIXCONRETRY** client environment variable
- SET ENVIRONMENT INFORMIXCONRETRY statement of SQL

The **INFORMIXCONTIME** setting takes precedence over the **INFORMIXCONRETRY** setting. Connection attempts can end after the **INFORMIXCONTIME** value is exceeded, but before the **INFORMIXCONRETRY** value is reached. For more information about restricting the time available to establish a connection to a database server, see INFORMIXCONTIME environment variable on page 1496

Related information

INFORMIXCONRETRY session environment option on page

INFORMIXCONRETRY configuration parameter on page

INFORMIXCONTIME environment variable

The **INFORMIXCONTIME** environment variable specifies the number of seconds the CONNECT statement attempts to establish a connection to a database server before returning an error. If you set no value, the default of 60 seconds can typically support a few hundred concurrent client connections. However, some systems might encounter few connection errors with a value as low as 15. The total distance between nodes, hardware speed, the volume of traffic, and the concurrency level of the network can all affect what value you should set to optimize **INFORMIXCONTIME**.

The **INFORMIXCONTIME** and **INFORMIXCONRETRY** environment variables let you configure your client-side connection capability to retry the connection instead of returning a **-908** error.

setenvINFORMIXCONTIME*seconds*

seconds

Represents the minimum number of seconds spent in attempts to establish a connection to a database server.

For example, enter this command to set **INFORMIXCONTIME** to `60` seconds:

```
setenv INFORMIXCONTIME 60
```

If **INFORMIXCONTIME** is set to `60` and **INFORMIXCONRETRY** is set to `3`, attempts to connect to the database server (after the initial attempt at `0` seconds) are made at `20`, `40`, and `60` seconds, if necessary, before aborting. This 20-second interval is the result of **INFORMIXCONTIME** divided by **INFORMIXCONRETRY**. If you set the **INFORMIXCONTIME** value to zero, the database server automatically uses the default value of 60 seconds.

If the CONNECT statement must search **DBPATH**, the **INFORMIXCONRETRY** setting specifies the number of additional connection attempts that can be made for each database server entry in **DBPATH**.

- All appropriate servers in the **DBPATH** setting are accessed at least once, even if the **INFORMIXCONTIME** value is exceeded. Thus, the CONNECT statement might take longer than the **INFORMIXCONTIME** time limit to return an error that indicates connection failure or that the database was not found.
- The **INFORMIXCONTIME** value is divided among the number of database server entries that are specified in **DBPATH**. Thus, if **DBPATH** contains numerous servers, increase the **INFORMIXCONTIME** value accordingly. For example, if **DBPATH** contains three entries, to spend at least `30` seconds attempting each connection, set **INFORMIXCONTIME** to `90`.

Order of precedence among INFORMIXCONTIME settings

When you specify a setting for the **INFORMIXCONTIME** client environment variable, it overrides the **INFORMIXCONTIME** configuration parameter settings in the `onconfig` file for the current session.

If the SET ENVIRONMENT statement specifies a setting for the INFORMIXCONRETRY session environment option, however, the SQL statement setting overrides the **INFORMIXCONRETRY** client environment variable setting for subsequent connection attempts during the current session. The SET ENVIRONMENT INFORMIXCONRETRY setting has no effect on other sessions.

In summary, this is the ascending order (lowest to highest) of the methods for setting an upper limit on the amount of time that a CONNECT statement can spend attempting to connect to a database server:

- **INFORMIXCONTIME** configuration parameter
- **INFORMIXCONTIME** client environment variable
- SET ENVIRONMENT INFORMIXCONTIME statement of SQL.

INFORMIXCONTIME takes precedence over the **INFORMIXCONRETRY** setting. Connection attempts can end after the **INFORMIXCONTIME** value is exceeded, but before the **INFORMIXCONRETRY** value is reached.

Related information

INFORMIXCONTIME session environment option on page

INFORMIXCONTIME configuration parameter on page

INFORMIXCPPMAP environment variable

Set the **INFORMIXCPPMAP** environment variable to specify the fully qualified pathname of the map file for C++ programs. Information in the map file includes the database server type, the name of the shared library that supports the database object or value object type, the library entry point for the object, and the C++ library for which an object was built.

```
setenvINFORMIXCPPMAPpathname
```

pathname

The directory path where the C++ map file is stored.

The map file is a text file that can have any filename. You can specify several map files, separated by colons (`:`) on UNIX™ or semicolons (`;`) on Windows™.

On UNIX™, the default map file is `$INFORMIXDIR/etc/c++map`. On Windows™, the default map file is `%INFORMIXDIR%\etc\c++map`.

INFORMIXDIR environment variable

The **INFORMIXDIR** environment variable specifies the directory that contains the subdirectories in which your product files are installed. You must always set **INFORMIXDIR**. Verify that **INFORMIXDIR** is set to the full pathname of the directory in which you installed your database server. If you have multiple versions of a database server, set **INFORMIXDIR** to the appropriate directory name for the version that you want to access. For information about when to set **INFORMIXDIR**, see your *HCL® Informix® Installation Guide*.

```
setenvINFORMIXDIR\pathname
```

pathname

is the directory path where the product files are installed.

To set **INFORMIXDIR** to **usr/informix/**, for example, as the installation directory, enter the following command:

```
setenv INFORMIXDIR /usr/informix
```

INFORMIXSERVER environment variable

The **INFORMIXSERVER** environment variable specifies the default database server to which an explicit or implicit connection is made by an SQL API client, the utility, or other HCL Informix® products.

This environment variable must be set before you can use HCL Informix® client products. It has the following syntax.

setenvINFORMIXSERVER*dbservername*

dbservername

is the name of the default database server.

The value of **INFORMIXSERVER** can be a local or remote server, but must correspond to a valid *dbservername* entry in the **$INFORMIXDIR/etc/sqlhosts** file on the computer running the application. The *dbservername* must begin with a lower-case letter and cannot exceed 128 bytes. It can include any printable characters except uppercase characters, field delimiters (blank space or tab), the newline character, and the hyphen (or minus) symbol.

For example, this command specifies the **coral** database server as the default:

```
setenv INFORMIXSERVER coral
```

INFORMIXSERVER specifies the database server to which an application connects if the CONNECT DEFAULT statement is executed. It also defines the database server to which an initial implicit connection is established if the first statement in an application is not a CONNECT statement.

Important: You must set **INFORMIXSERVER** even if the application or does not use implicit or explicit default connections.

INFORMIXSHMBASE environment variable (UNIX™)

The **INFORMIXSHMBASE** environment variable affects only client applications connected to HCL Informix® databases that use the interprocess communications (IPC) shared-memory (**ipcshm**) protocol.

Important: Resetting **INFORMIXSHMBASE** requires a thorough understanding of how the application uses memory. Normally you do not reset **INFORMIXSHMBASE**.

INFORMIXSHMBASE specifies where shared-memory communication segments are attached to the client process so that client applications can avoid collisions with other memory segments that it uses. If you do not set **INFORMIXSHMBASE**, the memory address of the communication segments defaults to an implementation-specific value such as `0x800000`.

setenvINFORMIXSHMBASE*value*

value

is an integer (in KB) used to calculate the memory address.

The database server calculates the memory address where segments are attached by multiplying the value of **INFORMIXSHMBASE** by `1,024`. For example, on a system that uses the C shell, you can set the memory address to the value `0x800000` by entering the following command:

```
setenv INFORMIXSHMBASE 8192
```

For more information, see your *HCL® Informix® Administrator's Guide* and the *HCL® Informix® Administrator's Reference*.

INFORMIXSQLHOSTS environment variable

The **INFORMIXSQLHOSTS** environment variable specifies where the SQL client or the database server can find connectivity information.

setenvINFORMIXSQLHOSTS*pathname*

pathname

The full path name of the connectivity information file.

UNIX: Default =`$INFORMIXDIR/etc/sqlhosts`

Windows server: Default = `%INFORMIXDIR%\etc\sqlhosts.%INFORMIXSERVER%`

For example, the following command overrides the default location and specifies that the `mysqlhosts` file is in the `/work/envt` directory:

```
setenv INFORMIXSQLHOSTS /work/envt/mysqlhosts
```

Windows™ client: The INFORMIXSQLHOSTS environment variable points to the computer whose registry contains the SQLHOSTS subkey. For example, the following command instructs the Windows™ client to look for connectivity information in the registry of a computer named **arizona**:

```
set INFORMIXSQLHOSTS = \\arizona
```

INFORMIXSTACKSIZE environment variable

The **INFORMIXSTACKSIZE** environment variable specifies the stack size (in KB) that is applied to all client processes. Any value that you set for INFORMIXSTACKSIZE in the client environment is ignored by the database server.

setenvINFORMIXSTACKSIZE*size*

size

is an integer, setting the stack size (in KB) for SQL client threads.

For example, to decrease the **INFORMIXSTACKSIZE** to 20 KB, enter the following command:

```
setenv INFORMIXSTACKSIZE 20
```

If **INFORMIXSTACKSIZE** is not set, the stack size is taken from the database server configuration parameter STACKSIZE or else defaults to a platform-specific value. The default stack size value for the primary thread of an SQL client is 32 KB for nonrecursive database activity.

Warning: For instructions on setting this value, see the *HCL® Informix® Administrator's Reference*. If you incorrectly set the value of **INFORMIXSTACKSIZE**, it can cause the database server to fail.

INFORMIXTERM environment variable (UNIX™)

The **INFORMIXTERM** environment variable specifies whether should use the information in the `terminfo` directory or the `termcap` file.

On character-based systems, the `terminfo` directory and `termcap` file determine terminal-dependent keyboard and screen capabilities, such as the operation of function keys, color and intensity attributes in screen displays, and the definition of window borders and graphic characters.

```
setenvINFORMIXTERM { terminfo  | termcap }
```

If **INFORMIXTERM** is not set, the default setting is `terminfo`.

The `terminfo` directory contains a file for each terminal name that has been defined. The `terminfo` setting for **INFORMIXTERM** is supported only on computers that provide full support for the UNIX™ System V `terminfo` library. For details, see the machine notes file for your product.

When is installed on your system, a `termcap` file is placed in the **etc** subdirectory of `$INFORMIXDIR`. This file is a superset of an operating-system **termcap** file. You can use the **termcap** file that the database server supplies, the system `termcap` file, or a `termcap` file that you create. You must set the **TERMCAP** environment variable if you do not use the default `termcap` file. For information about setting the **TERMCAP** environment variable, see TERMCAP environment variable (UNIX) on page 1514.

INF_ROLE_SEP environment variable

The **INF_ROLE_SEP** environment variable configures the security feature of role separation when the database server is installed or reinstalled on UNIX™ systems. Role separation enforces separating administrative tasks by people who run and audit the database server. After the installation is complete, **INF_ROLE_SEP** has no effect. If **INF_ROLE_SEP** is not set, then user **informix** (the default) can perform all administrative tasks.

```
setenvINF_ROLE_SEPn
```

n

is any positive integer.

On Windows™, the install process asks whether you want to enable role separation regardless of the setting of **INF_ROLE_SEP**. To enable role separation for database servers on Windows™, select the role-separation option during installation.

If **INF_ROLE_SEP** is set when HCL Informix® is installed on a UNIX™ platform, role separation is implemented and a separate group is specified to serve each of the following responsibilities:

- The Database Server Administrator (DBSA)
- The Audit Analysis Officer (AAO)
- The standard user

On UNIX™, you can establish role separation by changing the group that owns the `aaodir`, `dbsadir`, or `etc` directories at any time after the installation is complete. You can disable role separation by resetting the group that owns these directories to **informix**. You can have role separation enabled, for example, for the Audit Analysis Officer (AAO) without having role separation enabled for the Database Server Administrator (DBSA).

For more information about the security feature of role separation, see the *HCL® Informix® Security Guide*. To learn how to configure role separation when you install your database server, see your *HCL® Informix® Installation Guide*.

INTERACTIVE_DESKTOP_OFF environment variable (Windows™)

This environment variable lets you prevent interaction with the Windows™ desktop when an SPL routine executes a SYSTEM command.

```
setenvINTERACTIVE_DESKTOP_OFF { 1  | 0 }
```

If **INTERACTIVE_DESKTOP_OFF** is `1` and an SPL routine attempts to interact with the desktop (for example, with the `notepad.exe` or `cmd.exe` program), the routine fails unless the user is a member of the **Administrators** group.

The valid settings (`1` or `0`) have the following effects:

1

Prevents the database server from acquiring desktop resources for the user executing the stored procedure

0

SYSTEM commands in a stored procedure can interact with the desktop. This is the default value.

Setting **INTERACTIVE_DESKTOP_OFF** to `1` allows an SPL routine that does not interact with the desktop to execute more quickly. This setting also allows the database server to simultaneously call a greater number of SYSTEM commands because the command no longer depends on a limited operating- system resource (Desktop and WindowStation handles).

JAR_TEMP_PATH environment variable

Set the **JAR_TEMP_PATH** variable to specify a non-default local file system location where jar management procedures such as **install_jar( )** and **replace_jar( )** can store temporary **.jar** files of the Java™ virtual machine.

```
setenvJAR_TEMP_PATHpathname
```

pathname

specifies a local directory for temporary **.jar** files.

This directory must have read and write permissions for the user who starts the database server. If the **JAR_TEMP_PATH** environment variable is not set, temporary copies of **.jar** files are stored in the **/tmp** directory of the local file system for the database server.

JAVA_COMPILER environment variable

You can set the **JAVA_COMPILER** environment variable in the Java™ virtual machine environment to disable JIT compilation.

setenvJAVA_COMPILER { **none | NONE** }

The `NONE` and `none` settings are equivalent. On UNIX™ systems that support the C shell and on which **JAVA_COMPILER** has been set to `NONE` or `none`, you can enable the JIT compiler for the JVM environment by the following command:

```
unset JAVA_COMPILER
```

JVM_MAX_HEAP_SIZE environment variable

The **JVM_MAX_HEAP_SIZE** environment variable can set a non-default upper limit on the size of the heap for the Java™ virtual machine.

setenvJVM_MAX_HEAP_SIZE *size*

size

is a positive integer that specifies the maximum size (in megabytes).

For example, the following command sets the maximum heap size at 12 MB:

```
set JVM_MAX_HEAP_SIZE 12
```

If you do not set **JVM_MAX_HEAP_SIZE**, 16 MB is the default maximum size.

LD_LIBRARY_PATH environment variable (UNIX™)

The **LD_LIBRARY_PATH** environment variable tells the shell on Solaris systems which directories to search for client or shared HCL Informix® general libraries. You must specify the directory that contains your client libraries before you can use the product.

setenvLD_LIBRARY_PATH$PATH: *pathname*

pathname

Specifies the search path for the library.

For on AIX®, set **LIBPATH**. For on HP-UX, set **SHLIB_PATH**.

The following example sets the **LD_LIBRARY_PATH** environment variable to the directory:

```
setenv LD_LIBRARY_PATH
${INFORMIXDIR}/lib:${INFORMIXDIR}/lib/esql:$LD_LIBRARY_PATH
```

LIBPATH environment variable (UNIX™)

The **LIBPATH** environment variable tells the shell on AIX® systems which directories to search for dynamic-link libraries for the . You must specify the full path name for the directory where you installed the product.

```
setenvLIBPATH pathname
```

pathname

Specifies the search path for the libraries.

On Solaris, set **LD_LIBRARY_PATH**. On HP-UX, set **SHLIB_PATH**.

NODEFDAC environment variable

Enabling NODEFDAC applies the ANSI-compliant restrictions on default access privileges for the PUBLIC group when tables or Owner-mode user-defined routines are created in databases that are not ANSI-compliant.

In a database that is not ANSI-compliant, when the **NODEFDAC** environment variable enabled by setting it to `yes`,

- the database server withholds default table access privileges from PUBLIC when a new table is created,
- and also withholds the default Execute privilege from PUBLIC when an owner-privileged UDR is created.

```
setenvNODEFDAC { yes }
```

yes

prevents default table privileges (Select, Insert, Update, and Delete) from being granted to PUBLIC on new tables in a database that is not ANSI-compliant. This setting also prevents the Execute privilege from being granted to PUBLIC by default when a new user-defined routine is created in Owner mode.

The *yes* setting is case sensitive, and is also sensitive to leading and trailing blank spaces. Including uppercase letters or blank spaces in the setting is equivalent to leaving **NODEFDAC** unset. When **NODEFDAC** is not set, or if it is set to any value besides *yes*, default privileges on tables and Owner-mode UDRs are granted to PUBLIC by default when the table or UDR is created in a database that is not ANSI-compliant. The setting `YES`, for example, disables **NODEFDAC**.

Enabling **NODEFDAC** has no effect in an ANSI-compliant databases.

Important: Enabling **NODEFDAC** withholds default table or routine privileges from PUBLIC when the object is created, but the **NODEFDAC** setting cannot prevent the PUBLIC group from being granted the same privileges by a user who holds the necessary access privileges on the new table or on the new UDR.

ONCONFIG environment variable

The **ONCONFIG** environment variable specifies the name of the active file, called the `onconfig` file, which holds the configuration parameters for the database server.

This file is read as input during the initialization procedure. After you prepare your `onconfig` configuration file, set the **ONCONFIG** environment variable to the name of this file.

```
setenvONCONFIGfilename
```

filename

is the name of your `onconfig` file in the `%INFORMIXDIR%\etc\%ONCONFIG%` or `$INFORMIXDIR/etc/$ONCONFIG` directory

This file contains the configuration parameters for your database.

To prepare the `onconfig` file, make a copy of the `onconfig.std` file and modify the copy. Name the `onconfig` file so that it can easily be related to a specific database server. If you have multiple instances of a database server, each instance *must* have its own uniquely named `onconfig` file.

If the **ONCONFIG** environment variable is not set, the database server reads the configuration values from the `onconfig` file during initialization.

ONINIT_STDOUT environment variable (Windows™)

The **ONINIT_STDOUT** environment variable specifies a path and file name in which output from the oninit command is stored.

While it is not generally necessary to view output from the oninit command, it might be necessary in certain situations, such as when using the -v (verbose) option or when you want to see output from an unhandled exception in a process launched within a virtual processor. When the value of **ONINIT_STDOUT** is set to the name of a file, output from the oninit command is written to the file.

```
setONINIT_STDOUT\path\filename
```

You can set the **ONINIT_STDOUT** environment variable as a system variable in **Control Panel > System > Advanced > Environment Variables**. If the HCL Informix® service is configured to log on as user **informix**, start the service using the starts command after setting the environment variable. Note, however, that because environment variables are read from the system when the service is started, if the service is set to log on as the local system user, you must restart your computer for the environment variable to take effect. Because the local system user is effectively logged on at all times, environment variables are refreshed only when the operating system is restarted.

For example, if the environment variable set to `C:\temp\oninit_out.txt`, you can start the server with the verbose option with the following command:

```
starts %INFORMIXSERVER% -v
```

The oninit messages are saved to the `C:\temp\oninit_out.txt` file.

Important: Only a single instance of the database can run on a Windows™ machine if the **ONINIT_STDOUT** environment variable is set.

OPTCOMPIND environment variable

You can set the **OPTCOMPIND** environment variable so that the optimizer can select the appropriate join method.

```
setenvOPTCOMPIND{2 | 1 | 0}
```

0

A nested-loop join is preferred, where possible, over a sort-merge join or a hash join.

1

When the isolation level is *not* Repeatable Read, the optimizer behaves as in setting `2`; otherwise, the optimizer behaves as in setting `0`.

2

Nested-loop joins are not necessarily preferred. The optimizer bases its decision purely on costs, regardless of transaction isolation mode.

When **OPTCOMPIND** is not set, the database server uses the OPTCOMPIND value from the ONCONFIG configuration file. When neither the environment variable nor the configuration parameter is set, the default value is `2`.

On HCL Informix®, the SET ENVIRONMENT OPTCOMPIND statement can set or reset **OPTCOMPIND** dynamically at runtime. This overrides the current **OPTCOMPIND** value (or the ONCONFIG configuration parameter OPTCOMPIND) for the current user session only. For more information about the SET ENVIRONMENT OPCOMPIND statement of SQL see the *HCL® Informix® Guide to SQL: Syntax*.

For more information about the ONCONFIG configuration parameter OPTCOMPIND, see the *HCL® Informix® Administrator's Reference*. For more information about the different join methods that the optimizer uses, see your .

OPTMSG environment variable

Set the **OPTMSG** environment variable at runtime before you start the application to enable (or disable) optimized message transfers (message chaining) for all SQL statements in the application.

```
setenv OPTMSG{0 | 1}
```

0

disables optimized message transfers.

1

enables optimized message transfers and implements the feature for any subsequent connection.

The default value is `0` (zero), which explicitly disables message chaining. You might want, for example, to disable optimized message transfers for statements that require immediate replies, for debugging, or to ensure that the database server processes all messages before the application terminates.

When you set **OPTMSG** within an application, you can activate or deactivate optimized message transfers for each connection or within each thread. To enable optimized message transfers, you must set **OPTMSG** before you establish a connection.

For more information about setting **OPTMSG** and defining related global variables, see the *HCL® Informix® Enterprise Replication Guide*.

OPTOFC environment variable

Use the **OPTOFC** environment variable to enable optimize-OPEN-FETCH-CLOSE functionality in applications or other APIs (such as JDBC, ODBC, OLE DB, LIBDMI, and Lib C++) that use DECLARE and OPEN statements to establish a cursor.

```
setenvOPTOFC { 0  | 1 }
```

0

disables **OPTOFC** for all threads of the application.

1

enables **OPTOFC** for every cursor in every thread of the application.

The default value is `0` (zero).

You can set the **OPTOFC** environment variable on the client or server. If this environment variable is set on the server, then any application that does not explicitly set this environment variable uses the value that is set on the server.

The **OPTOFC** environment variable reduces the number of message requests between the application and the database server.

If you set **OPTOFC** from the shell, you must set it before you start the application. For more information about enabling **OPTOFC** and related features, see the *HCL® Informix® Enterprise Replication Guide*.

OPT_GOAL environment variable (UNIX™)

Set the **OPT_GOAL** environment variable in the user environment, before you start an application, to specify the query performance goal for the optimizer.

```
setenvOPT_GOAL { 0  | -1 }
```

0

Specifies user-response-time optimization.

-1

Specifies total-query-time optimization.

The default behavior is for the optimizer to use query plans that optimize the total query time.

You can also specify the optimization goal for individual queries with optimizer directives or for a session with the SET OPTIMIZATION statement.

Both methods take precedence over the **OPT_GOAL** environment variable setting. You can also set the OPT_GOAL configuration parameter for the HCL Informix® system; this method has the lowest level of precedence.

For more information about optimizing queries for your database server, see your . For information about the SET OPTIMIZATION statement, see the *HCL® Informix® Guide to SQL: Syntax*.

PATH environment variable

The UNIX™ **PATH** environment variable tells the shell which directories to search for executable programs. You must add the directory containing your HCL Informix® product to your **PATH** setting before you can use the product.

```
setenvPATH$PATH: pathname
```

pathname

Specifies the search path for the executable files.

Include a colon (:) separator between the path names on UNIX™ systems. (Use the semicolon (;) separator between path names on Windows™ systems.)

You can specify the search path in various ways. The **PATH** environment variable tells the operating system where to search for executable programs. You must include the directory that contains your HCL Informix® product in your **path** setting before you can use the product. This directory should be located before `$INFORMIXDIR/bin`, which you must also include.

For additional information about how to modify your path, see Modifying an environment-variable setting on page 1443.

PDQPRIORITY environment variable

The **PDQPRIORITY** environment variable determines the degree of parallelism that the database server uses and affects how the database server allocates resources, including memory, processors, and disk reads.

```
setenvPDQPRIORITY { HIGH  | LOW  | OFF | resources }
```

resources

Is an integer in the range `0` to `100`. The value `1` is the same as LOW, and `100` is the same as HIGH. Values lower than `0` are set to `0` (OFF), and values greater than `100` are set to `100` (HIGH).

Value `0` is the same as OFF (for HCL Informix® only).

Here the HIGH, LOW, and OFF keywords have the following effects:

HIGH

When the database server allocates resources among all users, it gives as many resources as possible to the query.

LOW

Data values are fetched from fragmented tables in parallel.

OFF

PDQ processing is turned off (for HCL Informix® only).

Usually, the more resources a database server uses, the better its performance for a given query. If the server uses too many resources, however, contention for the resources can take resources away from other queries, resulting in degraded performance. For more information about performance considerations for **PDQPRIORITY**, see the .

An application can override the setting of this environment variable when it issues the SQL statement SET PDQPRIORITY, as the *HCL® Informix® Guide to SQL: Syntax* describes.

Using PDQPRIORITY with Informix®

The *resources* value specifies the query priority level and the amount of resources that the database server uses to process the query.

When **PDQPRIORITY** is not set, the default value is `OFF`.

When **PDQPRIORITY** is set to HIGH, HCL Informix® determines an appropriate value to use for **PDQPRIORITY** based on several criteria. These include the number of available processors, the fragmentation of tables queried, the complexity of the query, and additional factors.

PSM_ACT_LOG environment variable

Use the **PSM_ACT_LOG** environment variable to specify the location of the HCL Informix® Primary Storage Manager activity log for your environment, for example, for a single session.

```
setenvPSM_ACT_LOG pathname
```

pathname

The full path name for the location of the `$INFORMIXDIR/psm_act.log`. If you specify a file name only, the storage manager creates the activity log in the working directory in which you started the storage manager.

The **PSM_ACT_LOG** environment variable overrides the value of the PSM_ACT_LOG configuration parameter.

Related information

PSM_ACT_LOG configuration parameter on page

PSM_CATALOG_PATH environment variable

Use the **PSM_CATALOG_PATH** environment variable to specify the location of the HCL Informix® Primary Storage Manager catalog tables for your environment, for example, for a single session.

```
setenvPSM_CATALOG_PATH pathname
```

pathname

The full path name for the location of the catalog table, which contain information about the pools, devices, and objects managed by the storage manager.

The **PSM_CATALOG_PATH** environment variable overrides the value of the PSM_CATALOG_PATH configuration parameter.

Related information

PSM_CATALOG_PATH configuration parameter on page

PSM_DBS_POOL environment variable

Use the **PSM_DBS_POOL** environment variable to change the name of the pool in which the HCL Informix® Primary Storage Manager places backup and restore dbspace data for your environment, for example, for a single session.

```
setenvPSM_DBS_POOL pool_name
```

pool_name

The name of the storage manager pool.

The **PSM_DBS_POOL** environment variable overrides the value of the PSM_DBS_POOL configuration parameter.

Related information

PSM_DBS_POOL configuration parameter on page

PSM_DEBUG environment variable

Use the **PSM_DEBUG** environment variable to specify the amount of debugging information that prints in the debug log for your environment, for example, for a single session.

```
setenvPSM_DEBUG value
```

value

`0` = No debugging messages.

`1` = Prints only internal errors.

`2` = Prints information about the entry and exit of functions and prints internal errors.

`3` = Prints the information specified by `1`-`2` with additional details.

`4` = Prints information about parallel operations and the information specified by `1`-`3`.

`5` = Prints information about internal states in the .

`6` = Prints the information specified by `1`-`5` with additional details.

`7` = Prints information specified by `1`-`6` with additional details.

`8` = Prints information specified by `1`-`7` with additional details.

`9` = Prints all debugging information.

The **PSM_DEBUG** environment variable overrides the value of the PSM_DEBUG configuration parameter.

Related information

PSM_DEBUG configuration parameter on page

PSM_DEBUG_LOG environment variable

Use the **PSM_DEBUG_LOG** environment variable to specify the location of the HCL Informix® Primary Storage Manager debug log for your environment, for example, for a single session.

```
setenvPSM_DEBUG_LOG pathname
```

pathname

The full path name for the location of the `$INFORMIXDIR/psm_debug.log`. If you specify a file name only, the storage manager creates the debug log in the working directory in which you started the storage manager.

The **PSM_DEBUG_LOG** environment variable overrides the value of the PSM_DEBUG_LOG configuration parameter.

Related information

PSM_DEBUG_LOG configuration parameter on page

PSM_LOG_POOL environment variable

Use the **PSM_LOG_POOL** environment variable to change the name of the pool in which the HCL Informix® Primary Storage Manager places backup and restore log data for your environment, for example, for a single session.

```
setenvPSM_LOG_POOL pool_name
```

pool_name

The name of the storage manager log pool.

The **PSM_LOG_POOL** environment variable overrides the value of the PSM_LOG_POOL configuration parameter.

Related information

PSM_LOG_POOL configuration parameter on page

PSORT_DBTEMP environment variable

The **PSORT_DBTEMP** environment variable specifies the location where the database server writes the temporary files that the **PSORT_NPROCS** environment variable uses to perform a sort.

```
setenvPSORT_DBTEMP pathname
```

pathname

The name of the UNIX™ directory used for intermediate writes during a sort.

To set the **PSORT_DBTEMP** environment variable to specify the directory (for example, `/usr/leif/tempsort`), enter the following command:

```
setenv PSORT_DBTEMP /usr/leif/tempsort
```

For maximum performance, specify directories that are located in file systems on different disks.

You might also want to consider setting the environment variable **DBSPACETEMP** to place temporary files used in sorting in dbspaces rather than operating-system files. See the discussion of the **DBSPACETEMP** environment variable in DBSPACETEMP environment variable on page 1477.

The database server uses the directory that **PSORT_DBTEMP** specifies, even if the environment variable **PSORT_NPROCS** is not set. For additional information about the **PSORT_DBTEMP** environment variable, see your *HCL® Informix® Administrator's Guide* and your .

PSORT_NPROCS environment variable

The **PSORT_NPROCS** environment variable enables the database server to improve the performance of the parallel-process sorting package by allocating more threads for sorting.

Before the sorting package performs a parallel sort, make sure that the database server has enough memory for the sort.

setenvPSORT_NPROCS*threads*

threads

is an integer, specifying the maximum number of threads to be used to sort a query. This value cannot be greater than `10`.

The following command sets **PSORT_NPROCS** to `4`:

```
setenv PSORT_NPROCS 4
```

To disable parallel sorting, enter the following command:

```
unsetenv PSORT_NPROCS
```

It is recommended that you initially set **PSORT_NPROCS** to `2` when your computer has multiple CPUs. If subsequent CPU activity is lower than I/O activity, you can increase the value of **PSORT_NPROCS**.

Tip: If the **PDQPRIORITY** environment variable is not set, the database server allocates the minimum amount of memory to sorting. This minimum memory is insufficient to start even two sort threads. If you have not set

PDQPRIORITY, check the available memory before you perform a large-scale sort (such as an index build) to make sure that you have enough memory.

Default PSORT_NPROCS values for detached indexes

If the **PSORT_NPROCS** environment variable is set, the database server uses the specified number of sort threads as an upper limit for ordinary sorts. If **PSORT_NPROCS** is not set, parallel sorting does not take place. If you have a single-CPU virtual processor, the database server uses one thread for the sort. If **PSORT_NPROCS** is set to `0`, the database server uses three threads for the sort.

Default PSORT_NPROCS values for attached indexes

The default number of threads is different for attached indexes.

If the **PSORT_NPROCS** environment variable is set, you get the specified number of sort threads for each fragment of the index that is being built.

If **PSORT_NPROCS** is not set, or if it is set to `0`, you get two sort threads for each fragment of the index unless you have a single-CPU virtual processor. If you have a single-CPU virtual processor, you get one sort thread for each fragment of the index.

For additional information about the **PSORT_NPROCS** environment variable, see your *HCL® Informix® Administrator's Guide* and your .

RTREE_COST_ADJUST_VALUE environment variable

The **RTREE_COST_ADJUST_VALUE** environment variable specifies a coefficient that support functions of user-defined data types can use to estimate the cost of an R-tree index for queries on UDT columns.

```
setenvRTREE_COST_ADJUST_VALUEvalue
```

value

is a floating-point number, where `1` ≤ *value* ≤ `1000`, specifying a multiplier for estimating the cost of using an index on a UDT column.

For spatial queries, the I/O overhead tends to exceed by far the CPU cost, so by multiplying the uncorrected estimated cost by an appropriate *value* from this setting, the database server can make better cost-based decisions on how to implement queries on UDT columns for which an R-tree index exists.

SRV_FET_BUF_SIZE environment variable

Use the **SRV_FET_BUF_SIZE** environment variable to specify the size of the fetch buffer that the local database server uses in distributed DML transactions across database servers.

```
setenvSRV_FET_BUF_SIZEsize
```

size

is a positive integer that is no greater than 1048576 (1 MiB), specifying the size (in bytes) of the fetch buffer that holds data retrieved by a cross-server distributed query.

For example, to set a buffer size to `5,000` bytes on a UNIX™ system that uses the C shell, set **SRV_FET_BUF_SIZE** by entering the following command:

```
setenv SRV_FET_BUF_SIZE 5000
```

When **SRV_FET_BUF_SIZE** is set to a valid value, the new value overrides the default value (or any previously set value) of **SRV_FET_BUF_SIZE**. The setting takes effect only when it is set in the starting environment of the database server.

When **SRV_FET_BUF_SIZE** is not set, the default setting for the fetch buffer is dependent on row size.

No error is raised if **SRV_FET_BUF_SIZE** is set to a value that is less than the default size, or that is greater than 1048576 (1MiB). If you specify a size for **SRV_FET_BUF_SIZE** that is greater than 1048576, the value is set to 1048576. In older 11.70 releases, up to and including 11.70.xC4, the upper limit is 32767.

A valid **SRV_FET_BUF_SIZE** setting is in effect only in cross-server DML transactions in which the local database server participates as the coordinator or as a subordinate database server.

- It has no effect, however, on queries that access only databases of the local server instance, and it does not affect the size of the fetch buffer in client-to-local-server communication.
- The processing of BYTE and TEXT objects is not affected by the **SRV_FET_BUF_SIZE** setting.
- Setting **SRV_FET_BUF_SIZE** for the environment of the local database server does not reset the fetch buffer size of remote server instances that coordinate or participate in cross-server DML transactions with the local server instance.

The greater the size of the buffer, the more rows can be returned, and the less frequently the local server must wait while the database server returns rows. A large buffer can improve performance when transferring a large amount of data between servers.

STMT_CACHE environment variable

Use the **STMT_CACHE** environment variable to control the use of the shared-statement cache on a session.

This feature can reduce memory consumption and can speed query processing among different user sessions. Valid **STMT_CACHE** values are `1` and `0`.

```
setenvSTMT_CACHE{1 | 0}
```

1

enables the SQL statement cache.

0

disables the SQL statement cache.

Set the **STMT_CACHE** environment variable for applications that do not use the SET STMT_CACHE statement to control the use of the SQL statement cache. By default, a statement cache is disabled, but can be enabled through the STMT_CACHE parameter of the `onconfig.std` file or by the SET STMT_CACHE statement.

This environment variable has no effect if the SQL statement cache is disabled through the configuration file setting. Values set by the SET STMT_CACHE statement in the application override the **STMT_CACHE** setting.

TERM environment variable (UNIX™)

The **TERM** environment variable is used for terminal handling. It lets (and other character-based applications) recognize and communicate with the terminal that you are using.

```
setenvTERMtype
```

type

Specifies the terminal type.

The terminal type specified in the **TERM** setting must correspond to an entry in the `termcap` file or `terminfo` directory.

Before you can set the **TERM** environment variable, you must obtain the code for your terminal from the database administrator.

For example, to specify the vt100 terminal, set the **TERM** environment variable by entering the following command:

```
setenv TERM vt100
```

TERMCAP environment variable (UNIX™)

The **TERMCAP** environment variable is used for terminal handling. It tells (and other character-based applications) to communicate with the `termcap` file instead of the `terminfo` directory.

```
setenvTERMCAPpathname
```

pathname

Specifies the location of the `termcap` file.

The `termcap` file contains a list of various types of terminals and their characteristics. For example, to provide terminal-handling information, which is specified in the `/usr/informix/etc/termcap` file, enter the following command:

```
setenv TERMCAP /usr/informix/etc/termcap
```

You can use set **TERMCAP** in any of the following ways. If several `termcap` files exist, they have the following (descending) order of precedence:

1. The `termcap` file that you create
2. The `termcap` file that the database server supplies (that is, `$INFORMIXDIR/etc/termcap`)
3. The operating-system `termcap` file (that is, `/etc/termcap`)

If you set the **TERMCAP** environment variable, be sure that the **INFORMIXTERM** environment variable is set to `termcap`.

If you do not set the **TERMCAP** environment variable, the `terminfo` directory is used by default.

TERMINFO environment variable (UNIX™)

The **TERMINFO** environment variable is used for terminal handling.

The environment variable is supported only on platforms that provide full support for the `terminfo` libraries that System V and Solaris UNIX™ systems provide.

```
setenvTERMINFO/usr/lib/terminfo
```

TERMINFO tells to communicate with the `terminfo` directory instead of the `termcap` file. The `terminfo` directory has subdirectories that contain files that pertain to terminals and their characteristics.

To set **TERMINFO**, enter the following command:

```
setenv    TERMINFO   /usr/lib/terminfo
```

THREADLIB environment variable (UNIX™)

Use the **THREADLIB** environment variable to compile multithreaded applications. A multithreaded application lets you establish as many connections to one or more databases as there are threads. These connections can remain active while the application program executes.

The **THREADLIB** environment variable indicates which thread package to use when you compile an application. Currently only the Distributed Computing Environment (DCE) is supported.

```
setenvTHREADLIBDCE
```

The **THREADLIB** environment variable is checked when the -thread option is passed to the script when you compile a multithreaded application. When you use the -thread option while compiling, the script generates an error if **THREADLIB** is not set, or if **THREADLIB** is set to an unsupported thread package.

TZ environment variable

The **TZ** environment variable is used for setting the time zone. It is used by various time functions to compute times relative to Coordinated Universal Time (UTC), formerly known as Greenwich Mean Time (GMT). The format is specified by the operating system.

```
setenvTZtzn [ { + | - } ] hh [ :mm [ :ss ] ] [ dzn ]
```

tzn

Three-letter time zone name, such as PST. You must specify the correct offset from local time to UTC (Universal Time Coordinated).

hh

A one- or two-digit difference in hours between UTC and local time. Optionally signed.

mm

Two-digit difference in minutes between UTC and local time.

ss

Two-digit difference in seconds between UTC and local time.

dzn

Three-letter daylight-saving-time zone, such as PDT. If daylight saving time is never in effect in the locality, set **TZ** without a value for *dzn*.

For example, if you use Pacific Standard Time with Pacific daylight savings time, set the **TZ** environment variable to `PST8PDT`. For more information on setting the **TZ** environment variable, see your operating system documentation.

USETABLENAME environment variable

The **USETABLENAME** environment variable can prevent users from using a synonym to specify the *table* in ALTER TABLE or DROP TABLE statements. Unlike most environment variables, **USETABLENAME** is not required to be set to a value. It takes effect if you set it to any value, or to no value.

```
setenvUSETABLENAME
```

By default, ALTER TABLE or DROP TABLE statements accept a valid synonym for the name of the *table* to be altered or dropped. (In contrast, RENAME TABLE issues an error if you specify a synonym, as do the ALTER SEQUENCE, DROP SEQUENCE, and RENAME SEQUENCE statements, if you attempt to substitute a synonym for the *sequence* name in those statements.)

If you set **USETABLENAME**, an error results if a synonym is in ALTER TABLE or DROP TABLE statements. Setting **USETABLENAME** has no effect on the DROP VIEW statement, which accepts a valid synonym for the view.

Appendixes

The stores_demo Database

The **stores_demo** database contains a set of tables that describe an imaginary business and many of the examples in the HCL Informix® documentation are based on this database.

The **stores_demo** database uses the default (U.S. English) locale and is not ANSI-compliant.

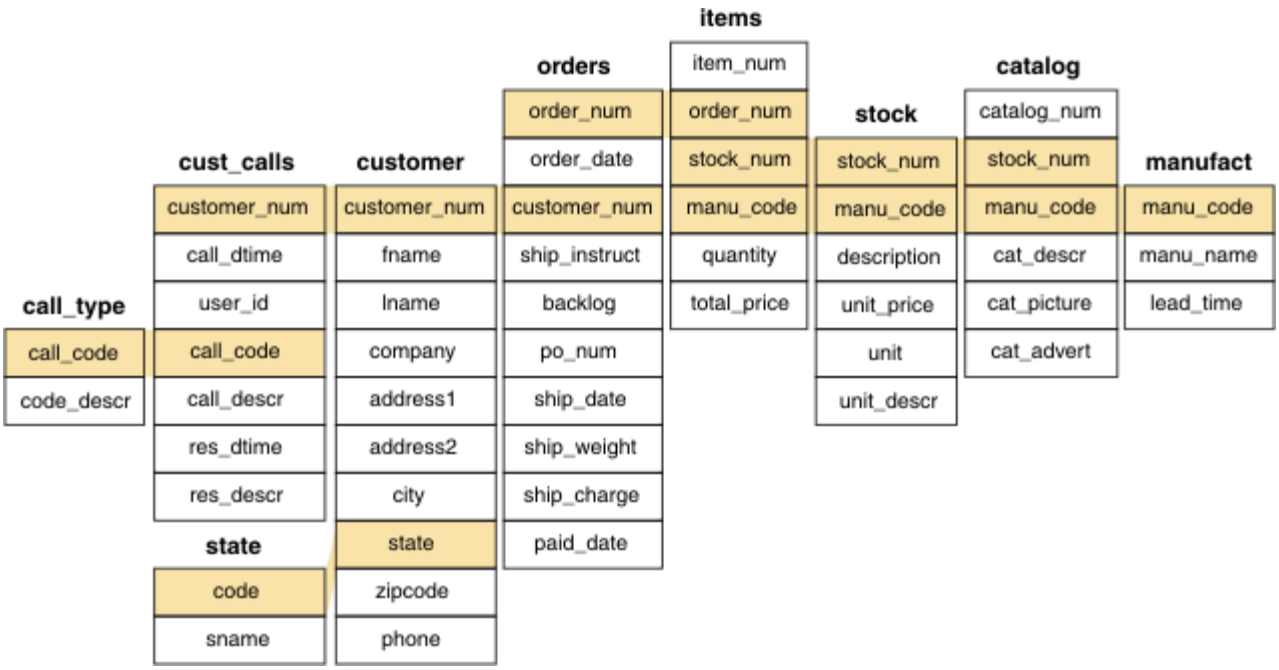
For information about how to create and populate the **stores_demo** database, see the *HCL® Informix® DB-Access User's Guide*. For information about how to design and implement a relational database, see the *IBM® Informix® Database Design and Implementation Guide*.

The stores_demo Database Map

Some of the tables in the **stores_demo** database have relationships between them.

The following illustration displays the joins in the **stores_demo** database between customers, catalog orders, and customer calls. The shading that connects a column in one table to a column with the same name in another table indicates the relationships, or *joins*, between tables.

Figure 19. Joins between customers and catalog orders



The **stores_demo** database also contains tables that hold electricity meter data fro customers. The **Customer_ts_data** and **ts_data** tables contain time series data. The **customer** table and the **Customer_ts_data** table are joined by the **customer_num** column. The **Customer_ts_data** table is joined to the **ts_data** table by the **loc_esi_id**, **measure_unit**, and **direction** columns. You can prevent the creation of these time series tables when you create the demonstration database.

The superstores_demo database

The **superstores_demo** database illustrates an object-relational schema.

SQL files and user-defined routines (UDRs) that are provided with DB-Access let you derive the **superstores_demo** object-relational database.

The **superstores_demo** database uses the default locale and is not ANSI-compliant.

For information about how to create and populate the demonstration databases, including relevant SQL files, see the *HCL® Informix® DB-Access User's Guide*. For conceptual information about demonstration databases, see the *IBM® Informix® Database Design and Implementation Guide*.

Structure of the superstores_demo Tables

Although many of the tables in the **superstores_demo** database have the same name as **stores_demo** tables, they are different.

The **superstores_demo** database includes the following tables. The tables are listed alphabetically, not in the order in which they are created.

- **call_type**
- **catalog**
- **cust_calls**
- **customer**
    - **retail_customer**
    - **whlsale_customer**
- **items**
- **location**
    - **location_non_us**
    - **location_us**
- **manufact**
- **orders**
- **region**
- **sales_rep**
- **state**
- **stock**
- **stock_discount**
- **units**

User-defined routines and extended data types

The **superstores_demo** database uses user-defined routines (UDRs) and extended data types.

A UDR is a routine that you define that can be invoked within an SQL statement or another UDR. A UDR can either return values or not.

The data type system of HCL Informix® is an extensible and flexible system that supports the creation of following kinds of data types:

- Extensions of existing data types by, redefining some of the behavior for data types that the database server provides
- Definitions of customized data types by a user

For information about creating and using UDRs and extended data types, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

The **superstores_demo** database creates the *distinct* data type, percent, in a UDR, as follows:

```
CREATE DISTINCT TYPE percent AS DECIMAL(5,5);
DROP CAST (DECIMAL(5,5) AS percent);
CREATE IMPLICIT CAST (DECIMAL(5,5) AS percent);
The superstores_demo database creates the following named row types:
```

- **location** hierarchy:
  - **location_t**
  - **loc_us_t**
  - **loc_non_us_t**
- **customer** hierarchy:
  - **name_t**
  - **customer_t**
  - **retail_t**
  - **whlsale_t**
- **orders** table
  - **ship_t**

location_t definition

```
location_id              SERIAL
loc_type                 CHAR(2)
company                  VARCHAR(20)
street_addr              LIST(VARCHAR(25) NOT NULL)
city                     VARCHAR(25)
country                  VARCHAR(25)
```

loc_us_t definition

```
state_code              CHAR(2)
zip                     ROW(code INTEGER, suffix SMALLINT)
phone                   CHAR(18)
```

loc_non_us_t definition

```
province_code           CHAR(2)
zipcode                 CHAR(9)
phone                   CHAR(15)
```

name_t definition

```
first               VARCHAR(15)
last                VARCHAR(15)
```

customer_t definition

```
customer_num                SERIAL
customer_type               CHAR(1)
customer_name               name_t
customer_loc                INTEGER
contact_dates               LIST(DATETIME YEAR TO DAY NOT NULL)
cust_discount               percent
credit_status               CHAR(1)
```

retail_t definition

```
credit_num              CHAR(19)
expiration              DATE
```

whlsale_t definition

```
resale_license          CHAR(15)
terms_net               SMALLINT
```

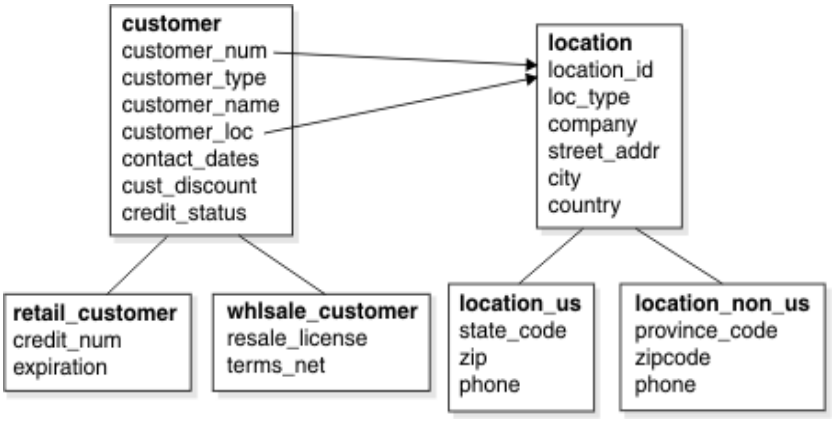
ship_t definition

```
date                    DATE
weight                  DECIMAL(8,2)
charge                  MONEY(6,2)
instruct                VARCHAR(40)
```

Table Hierarchies

The following illustration shows how the hierarchical tables of the **superstores_demo** database are related. The foreign key and primary relationships between the two tables are indicated by shaded arrows that point from the **customer.custnum** and **customer.loc** columns to the **location.location_id** columns.

Figure 20. Hierarchies of superstores_demo Tables



Guide to SQL: Tutorial

The *HCL® Informix® Guide to SQL: Tutorial* shows how to use basic and advanced structured query language (SQL) to access and manipulate the data in your databases. It discusses the data manipulation language (DML) statements as well as triggers and stored procedure language (SPL) routines, which DML statements often use.

This publication is written for the following users:

- Database users
- Database administrators
- Database-application programmers

This publication assumes that you have the following background:

- A working knowledge of your computer, your operating system, and the utilities that your operating system provides
- Some experience working with relational databases or exposure to database concepts
- Some experience with computer programming

This publication is one of a series of publications that discusses the HCL® Informix® implementation of SQL. The *HCL® Informix® Guide to SQL: Syntax* contains all the syntax descriptions for SQL and SPL. The *HCL® Informix® Guide to SQL: Reference* provides reference information for aspects of SQL other than the language statements. The *IBM® Informix® Database Design and Implementation Guide* shows how to use SQL to implement and manage your databases.

Database concepts

This chapter describes fundamental database concepts and focuses on the following topics:

- Data models
- Multiple users
- Database terminology
- SQL (Structured Query Language)

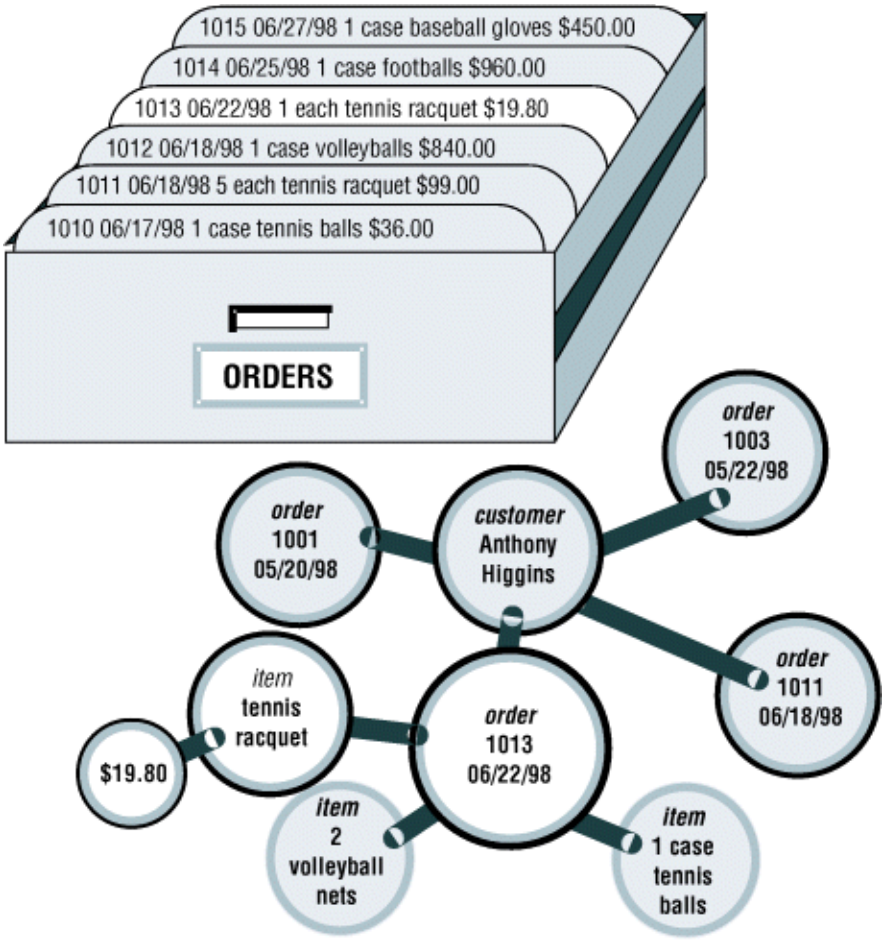
Your real use of a database begins with the SELECT statement, which Compose SELECT statements on page 1532, describes.

Illustration of a data model

The principal difference between information collected in a database versus information collected in a file is the way the data is organized. A flat file is organized physically; certain items precede or follow other items. But the contents of a database are organized according to a *data model*. A data model is a plan, or map, that defines the units of data and specifies how each unit relates to the others.

For example, a number can appear in either a file or a database. In a file, it is simply a number that occurs at a certain point in the file. A number in a database, however, has a role that the data model assigns to it. The role might be a *price* that is associated with a *product* that was sold as one *item* of an *order* that a *customer* placed. Each of these components, price, product, item, order, and customer, also has a role that the data model specifies. For an illustration of a data model, see the following figure.

Figure 21. The advantage of using a data model



You design the data model when you create the database. You then insert units of data according to the plan that the model lays out. Some books use the term *schema* instead of *data model*.

Store data

Another difference between a database and a file is that the organization of the database is stored with the database.

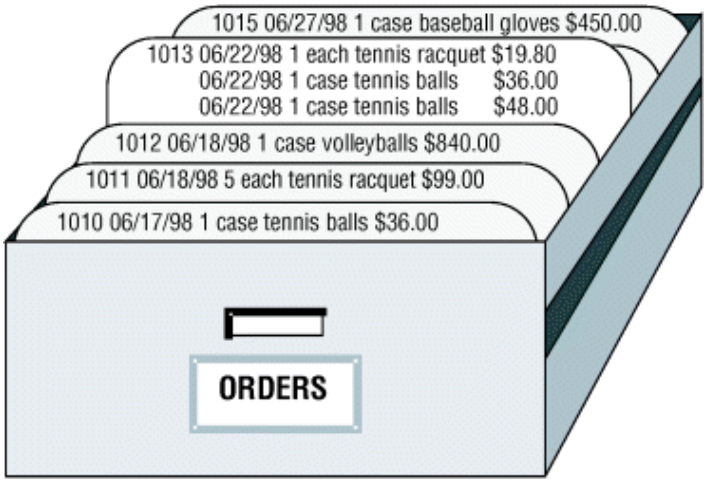
A file can have a complex inner structure, but the definition of that structure is not within the file; it is in the programs that create or use the file. For example, a document file that a word-processing program stores might contain detailed structures that describe the format of the document. However, only the word-processing program can decipher the contents of the file, because the structure is defined within the program, not within the file.

A data model, however, is contained in the database it describes. It travels with the database and is available to any program that uses the database. The model defines not only the names of the data items but also their data types, so a program can adapt itself to the database. For example, a program can find out that, in the current database, a price item is a decimal number with eight digits, two to the right of the decimal point; then it can allocate storage for a number of that type. How programs work with databases is the subject of SQL programming on page 1700, and Modify data through SQL programs on page 1723.

Query data

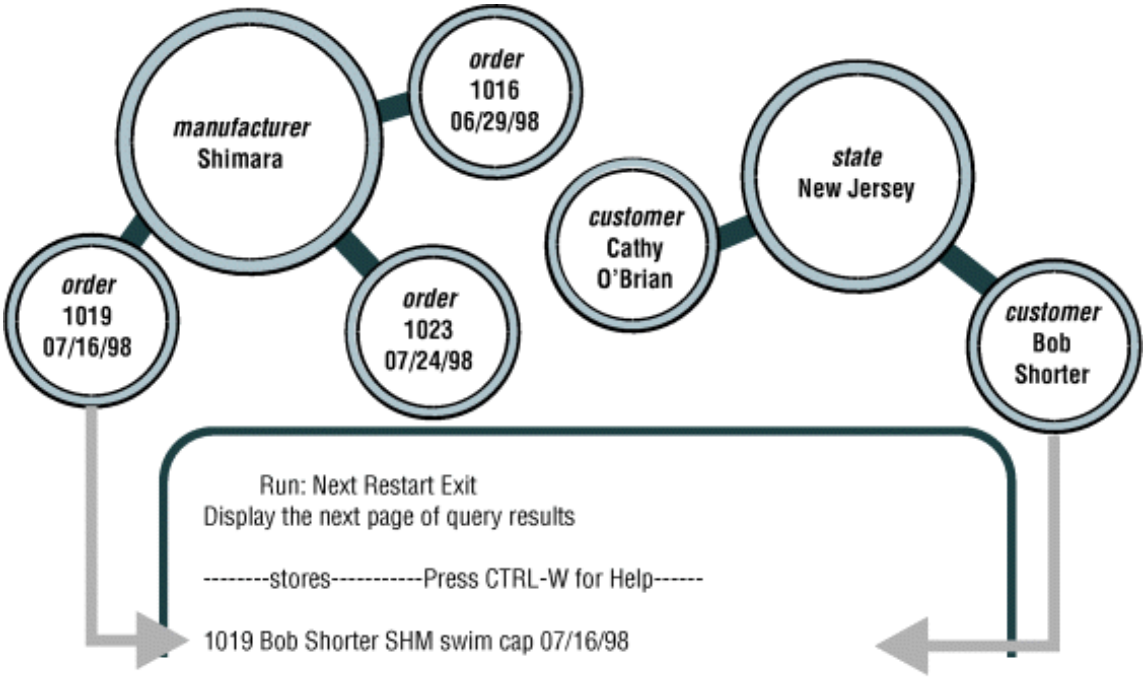
Another difference between a database and a file is the way you can access them. You can search a file sequentially, looking for particular values at particular physical locations in each line or record. That is, you might ask, *"What records have the number 1013 in the first field?"* The following figure shows this type of search.

Figure 22. Search a file sequentially



In contrast, when you query a database, you use the terms that the model defines. You can query the database with questions such as, *"What orders have been placed for products made by the Shimara Corporation, by customers in New Jersey, with ship dates in the third quarter?"* The following figure shows this type of query.

Figure 23. Query a database



In other words, when you access data that is stored in a file, you must state your question in terms of the physical layout of the file. When you query a database, you can ignore the arcane details of computer storage and state your query in terms that reflect the real world, at least to the extent that the data model reflects the real world.

Compose SELECT statements on page 1532, and Compose advanced SELECT statements on page 1619, discuss the language you use to make queries.

For information about how to build and implement your data model, see the *IBM® Informix® Database Design and Implementation Guide*.

Modify data

The data model also makes it possible to modify the contents of the database with less chance for error. You can query the database with statements such as, *"Find every stock item with a manufacturer of Presta or Schraeder, and increase its price by 13 percent."* You state changes in terms that reflect the meaning of the data. You do not have to waste time and effort thinking about details of fields within records in a file, so the chances for error are fewer.

The statements you use to modify stored data are covered in Modify data on page 1658.

Concurrent use and security

A database can be a common resource for many users. Multiple users can query and modify a database simultaneously. The database server (the program that manages the contents of all databases) ensures that the queries and modifications are done in sequence and without conflict.

Having concurrent users on a database provides great advantages but also introduces new problems of security and privacy. Some databases are private; individuals set them up for their own use. Other databases contain confidential material that must be shared, but only among a restricted group; still other databases provide public access.

Control database use

HCL® Informix® database software provides the means to control database use. When you design a database, you can perform any of the following functions:

- Keep the database completely private
- Open its entire contents to all users or to selected users
- Restrict the selection of data that some users can view (different selections of data to different groups of users)
- Allow specified users to view certain items, but not modify them
- Allow specified users to add new data, but not modify old data
- Allow specified users to modify all, or specified items of, existing data
- Ensure that added or modified data conforms to the data model

Access-management strategies

HCL Informix® supports two access-management systems:

Label-Based Access Control (LBAC)

Label-Based Access Control is an implementation of Mandatory Access Control, which is typically used in databases that store highly sensitive data, such as systems maintained by armed forces or security services. The primary documentation of features relating to LBAC is the *HCL® Informix® Security Guide*. *HCL® Informix® Guide to SQL: Syntax* describes how LBAC security objects are created and maintained by the Database Security Administrator (DBSECADM). Only the Database Server Administrator (DBSA) can grant the DBSECADM role.

Discretionary Access Control (DAC)

Discretionary Access Control is a simpler system that involves less overhead than LBAC. Based on access privileges and roles, DAC is enabled in all Informix® databases, including those that implement LBAC.

Creating and granting a role

About this task

To support DAC, the database administrator (DBA) can define *roles* and assign them to users to standardize the access privileges of groups of users who need access to the same database objects. When the DBA assigns privileges to that role, every user who is granted role holds those privileges when that role is activated. In order to activate a specific role, a user must issue the SET ROLE statement. The SQL statements used for defining and manipulating roles include: CREATE ROLE, DROP ROLE, GRANT, REVOKE, and SET ROLE.

For more information on the SQL syntax statements for defining and manipulating roles, see the *HCL® Informix® Guide to SQL: Syntax*.

To create and grant a role:

1. Use the CREATE ROLE statement to create a new role in the current database.
2. Use the GRANT statement to grant access privileges to that role
3. Use the GRANT statement to grant the role to a user or to PUBLIC (all users).
4. The user must issue the SET ROLE statement to enable that role.

Defining and granting privileges for a default role

About this task

The DBA can also define a *default role* to assign to individual users or to the PUBLIC group for a specific database. The role is automatically activated when the user establishes a connection with that database, without the requiring the user to issue a SET ROLE statement. At connection time, each user who holds a default role has whatever access privileges are granted to the user individually, as well as the privileges of the default role.

Only one role that the CREATE ROLE statement defines can be in effect for a given user at a given time. If a user who holds both a default role and one or more other roles uses the SET ROLE statement to make a nondefault role the active role, then any access privileges that were granted only to the default role (and not to the user individually, nor to PUBLIC, nor to the new

active role) are no longer in effect for that user. The same user can issue the SET ROLE DEFAULT statement to reactivate the default role, but this action disables any privileges that the user held only through the previously enabled nondefault role.

If different default roles are assigned to the user and to PUBLIC, the default role of the user takes precedence.

To define and grant privileges for a default role:

1. Use the CREATE ROLE statement to create a new role in the current database.
2. Use the GRANT statement to grant privileges to the role.
3. Grant the role to a user and set the role as the default user or PUBLIC role using the one of the following syntax:
   Choose from:
   - `GRANT DEFAULT ROLE rolename TO username;`
   - `GRANT DEFAULT ROLE rolename TO PUBLIC;`
4. Use the REVOKE DEFAULT ROLE statement to disassociate a default role from a user.

   Restriction: Only the DBA or the database owner can remove the default role.

5. Use the SET ROLE DEFAULT statement to reset the current role back to the default role.

Built-in roles

For security reasons, HCL Informix® supports certain built-in roles that are in effect for any user who is granted the role and is connected to the database, regardless of whether any other role is also active.

For example, in a database in which the IFX_EXTEND_ROLE configuration parameter is set to ON, only the Database Server Administrator (DBSA) or users to whom the DBSA has granted the built-in EXTEND role can create or drop UDRs that are defined with the EXTERNAL keyword.

Similarly, in a database that implements LBAC security policies, the DBSA can grant the built-in DBSECADM role. The grantee of this role becomes the Database Security Administrator, who can define and implement LBAC security policies and can assign security labels to data and to users.

Unlike user-defined roles, built-in roles cannot be destroyed by the DROP ROLE statement. The SET ROLE statement has no effect on a built-in role, because it is always active while users are connected to a database in which they have been granted the built-in role.

For more information on the External Routine Reference segment or SQL statements for defining and manipulating roles, see the *HCL® Informix® Guide to SQL: Syntax*.

For more information on the DBSECADM role or SQL statements for defining and manipulating LBAC security objects, see the *HCL® Informix® Security Guide*.

For more information on default roles, see the *HCL® Informix® Administrator's Guide*.

For more information about how to grant and limit access to your database, see the *IBM® Informix® Database Design and Implementation Guide*.

Centralized management

Databases that many people use are valuable and must be protected as important business assets. You create a significant problem when you compile a store of valuable data and simultaneously allow many employees to access it. You handle this problem by protecting data while maintaining performance. The database server lets you centralize these tasks.

Databases must be guarded against loss or damage. The hazards are many: failures in software and hardware, and the risks of fire, flood, and other natural disasters. Losing an important database creates a huge potential for damage. The damage could include not only the expense and difficulty of re-creating the lost data, but also the loss of productive time by the database users as well as the loss of business and goodwill while users cannot work. A plan for regular backups helps avoid or mitigate these potential disasters.

A large database that many people use must be maintained and tuned. Someone must monitor its use of system resources, chart its growth, anticipate bottlenecks, and plan for expansion. Users will report problems in the application programs; someone must diagnose these problems and correct them. If rapid response is important, someone must analyze the performance of the system and find the causes of slow responses.

Important database terms

You should know a number of terms before you begin the next chapter. Depending on the database server you use, a different set of terms can describe the database and the data model that apply.

The relational database model

The databases you create with the HCL® Informix® database server are *object-relational* databases. In practical terms this means that all data is presented in the form of *tables* with *rows* and *columns* where the following simple corresponding relationships apply.

Relationship

Description

table = entity

A table represents all that the database knows about one subject or kind of thing.

column = attribute

A column represents one feature, characteristic, or fact that is true of the table subject.

row = instance

A row represents one individual instance of the table subject.

Some rules apply about how you choose entities and attributes, but they are important only when you are designing a new database. (For more information about database design, see the *IBM® Informix® Database Design and Implementation Guide*.) The data model in an existing database is already set. To use the database, you need to know only the names of the tables and columns and how they correspond to the real world.

Tables

A *database* is a collection of information that is grouped into one or more tables. A table is an array of data *items* organized into rows and columns. A demonstration database is distributed with every HCL® Informix® database server product. A partial table from the demonstration database follows.

stock_num	manu_code	description	unit_price	unit	unit_descr
. . .	. . .	. . .	. . .	. . .	. . .
1	HRO	baseball gloves	250.00	case	10 gloves/case
1	HSK	baseball gloves	800.00	case	10 gloves/case
1	SMT	baseball gloves	450.00	case	10 gloves/case
2	HRO	baseball	126.00	case	24/case
3	HSK	baseball bat	240.00	case	12/case
4	HSK	football	960.00	case	24/case
4	HRO	football	480.00	case	24/case
5	NRG	tennis racquet	28.00	each	each
. . .	. . .	. . .	. . .	. . .	. . .
313	ANZ	swim cap	60.00	case	12/box

A table represents all that the database administrator (DBA) wants to store about one *entity*, one type of thing that the database describes. The example table, **stock**, represents all that the DBA wants to store about the merchandise that a sporting goods store stocks. Other tables in the demonstration database represent such entities as **customer** and **orders**.

Columns

Each column of a table contains one *attribute*, which is one characteristic, feature, or fact that describes the subject of the table. The **stock** table has columns for the following facts about items of merchandise: stock numbers, manufacturer codes, descriptions, prices, and units of measure.

Rows

Each row of a table is one *instance* of the subject of the table, which is one particular example of that entity. Each row of the **stock** table stands for one item of merchandise that the sporting goods store sells.

Views

A *view* is a virtual table based on a specified SELECT statement. A view is a dynamically controlled picture of the contents in a database and allows a programmer to determine what information the user sees and manipulates. Different users can be given different views of the contents of a database, and their access to those contents can be restricted in several ways.

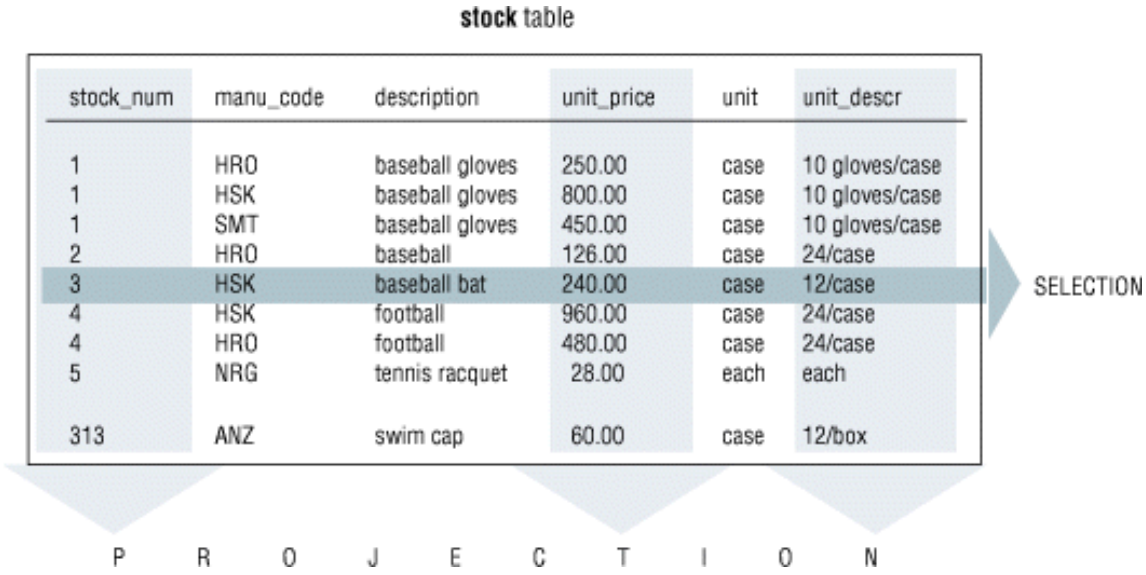
Sequences

A *sequence* is a database object that generates a sequence of whole numbers within a defined range. The sequence of numbers can run in either ascending or descending order, and is monotonic. For more information about sequences, see the *HCL® Informix® Guide to SQL: Syntax*.

Operations on tables

Because a database is really a collection of tables, database operations are operations on tables. The object-relational model supports three fundamental operations: selection, projection, and joining. The following figure shows the selection and projection operations. (All three operations are defined in detail, with many examples, in the following topics.)

Figure 24. Illustration of selection and projection



When you *select* data from a table, you are choosing certain rows and ignoring others. For example, you can query the **stock** table by asking the database management system to, *"Select all rows in which the manufacturer code is HSK and the unit price is between 200.00 and 300.00."*

When you *project* from a table, you are choosing certain columns and ignoring others. For example, you can query the **stock** table by asking the database management system to *"project the **stock_num**, **unit_descr**, and **unit_price** columns."*

A table contains information about only one entity; when you want information about multiple entities, you must *join* their tables. You can join tables in many ways. For more information about join operations, refer to Compose advanced SELECT statements on page 1619.

The object-relational model

HCL Informix® (Informix®) allows you to build *object-relational* databases. In addition to supporting alphanumeric data such as character strings, integers, date, and decimal, an object-relational database extends the features of a relational model with the following object-oriented capabilities:

Extensibility

You can extend the capability of the database server by defining new data types (and the access methods and functions to support them) and user-defined routines (UDRs) that allow you to store and manage images, audio, video, large text documents, and so forth.

IBM®, as well as third-party vendors, packages some data types and access methods into modules or shared class libraries, that you can add on to the database server, if it suits your needs. A module enables you to store non-traditional data types such as two-dimensional spatial objects (lines, polygons, ellipses, and circles) and to access them through R-tree indexes. A module might also provide new types of access to large text documents, including phrase matching, fuzzy searches, and synonym matching.

You can also extend the database server on your own by using the features of that enable you to add data types and access methods. For more information, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

You can create UDRs in SPL and the C programming language to encapsulate application logic or to enhance the functionality of the Informix®. For more information, see Create and use SPL routines on page 1753.

Complex types

You can define new data types that combine one or more existing data types. Complex types enable greater flexibility in organizing data at the level of columns and tables. For example, with complex types, you can define columns that contain collections of values of a single type and columns that contain multiple component types.

Inheritance

You can define objects (types and tables) that acquire the properties of other objects and add new properties that are specific to the object that you define.

Informix® provides object-oriented capabilities beyond those of the relational model but represents all data in the form of tables with rows and columns. Although the object-relational model extends the capabilities of the relational model, you can implement your data model as a traditional relational database if you choose.

Some rules apply about how you choose entities and attributes, but they are important only when you are designing a new database. For more information about object-relational database design, see the *IBM® Informix® Database Design and Implementation Guide*.

Structured Query Language

Most computer software has not yet reached a point where you can literally ask a database, *"What orders have been placed by customers in New Jersey with ship dates in the third quarter?"* You must still phrase questions in a restricted syntax that the software can easily parse. You can pose the same question to the demonstration database in the following terms:

```
SELECT * FROM customer, orders
   WHERE customer.customer_num = orders.customer_num
      AND customer.state = 'NJ'
```

```
    AND orders.ship_date
    BETWEEN DATE('7/1/98') AND DATE('9/30/98');
```

This question is a sample of Structured Query Language (SQL). It is the language that you use to direct all operations on the database. SQL is composed of statements, each of which begins with one or two keywords that specify a function. The HCL® Informix® implementation of SQL includes a large number of SQL statements, from ALLOCATE DESCRIPTOR to WHENEVER.

You will use most of the statements only when you set up or tune your database. You will use three or four statements regularly to query or update your database. For details on SQL statements, see the *HCL® Informix® Guide to SQL: Syntax*.

One statement, SELECT, is in almost constant use. SELECT is the only statement that you can use to retrieve data from the database. It is also the most complicated statement, and the next two chapters of this book explore its many uses.

General programming

You can write programs that incorporate SQL statements and exchange data with the database server. That is, you can write a program to retrieve data from the database and format it however you choose. You can also write programs that take data from any source in any format, prepare it, and insert it into the database.

You can also write programs called stored routines to work with database data and objects. The stored routines that you write are stored directly in a database in tables. You can then execute a stored routine from or an SQL Application Programming Interface (API) such as .

SQL programming on page 1700, and Modify data through SQL programs on page 1723, present an overview of how SQL is used in programs.

ANSI-compliant databases

Use the MODE ANSI keywords when you create a database to designate it as ANSI compliant. Within such a database, certain characteristics of the ANSI/ISO standard apply. For example, all actions that modify data take place within a transaction automatically, which means that the changes are made in their entirety or not at all. Differences in the behavior of ANSI-compliant databases are noted, where appropriate, in the statement descriptions in the *HCL® Informix® Guide to SQL: Syntax*. For a detailed discussion of ANSI-compliant databases, see the *IBM® Informix® Database Design and Implementation Guide*.

Global Language Support

HCL® Informix® database server products provide the Global Language Support (GLS) feature. In addition to U.S. ASCII English, GLS allows you to work in other locales and use non-ASCII characters in SQL data and identifiers. You can use the GLS feature to conform to the customs of a specific locale. The locale files contain culture-specific information, such as money and date formats and collation orders. For more GLS information, see the *HCL® Informix® GLS User's Guide*.

Summary

A database contains a collection of related information but differs in a fundamental way from other methods of storing data. The database contains not only the data, but also a data model that defines each data item and specifies its meaning with respect to the other items and to the real world.

More than one user can access and modify a database at the same time. Each user has a different view of the contents of a database, and each user's access to those contents can be restricted in several ways.

A relational database consists of tables, and the tables consist of columns and rows. The relational model supports three fundamental operations on tables: selections, projections, and joins.

An object-relational database extends the features of a relational database. You can define new data types to store and manage audio, video, large text documents, and so forth. You can define complex types that combine one or more existing data types to provide greater flexibility in how you organize your data in columns and tables. You can define types and tables that inherit the properties of other database objects and add new properties that are specific to the object that you define.

To manipulate and query a database, use SQL. IBM® pioneered SQL and ANSI standardized it. HCL® Informix® extensions that you can use to your advantage add to the ANSI-defined language. HCL® Informix® tools also make it possible to maintain strict compliance with ANSI standards.

Two layers of software mediate all your work with databases. The bottom layer is always a database server that executes SQL statements and manages the data on disk and in computer memory. The top layer is one of many applications, some from IBM® and some written by you, by other vendors, or your colleagues. Middleware is the component that links the database server to the application, and is provided by the database vendor to bind the client programs with the database server. HCL® Informix® Stored Procedure Language (SPL) is an example of such a tool.

Compose SELECT statements

The SELECT statement is the most important and the most complex SQL statement. You can use it and the SQL statements INSERT, UPDATE, and DELETE to manipulate data. You can use the SELECT statement to retrieve data from a database, as part of an INSERT statement to produce new rows, or as part of an UPDATE statement to update information.

The SELECT statement is the primary way to query information in a database. It is your key to retrieving data in a program, report, form, or spreadsheet. You can use SELECT statements with a query tool such as or embed SELECT statements in an application.

This chapter introduces the basic methods for using the SELECT statement to query and retrieve data from relational databases. It discusses how to tailor your statements to select columns or rows of information from one or more tables, how to include expressions and functions in SELECT statements, and how to create various join conditions between database tables. The syntax and usage for the SELECT statement are described in detail in the *HCL® Informix® Guide to SQL: Syntax*.

Most examples in this publication come from the tables in the **stores_demo** database, which is included with the software for your Informix® SQL API or database utility. In the interest of brevity, the examples show only part of the data that is retrieved for each SELECT statement. For information on the structure and contents of the demonstration database, see

the *HCL® Informix® Guide to SQL: Reference*. For emphasis, keywords are shown in uppercase letters in the examples, although SQL is not case sensitive.

SELECT statement overview

The SELECT statement is constructed of clauses that let you look at data in a relational database. These clauses let you select columns and rows from one or more database tables or views, specify one or more conditions, order and summarize the data, and put the selected data in a temporary table.

This chapter shows how to use five SELECT statement clauses. If you include all five of these clauses, they must appear in the SELECT statement in the following order:

1. Projection clause
2. FROM clause
3. WHERE clause
4. ORDER BY clause
5. INTO TEMP clause

Only the Projection clause and FROM clause are required. These two clauses form the basis for every database query, because they specify the column values to be retrieved, and the tables that contain those columns. Use one or more of the other clauses from the following list:

- Add a WHERE clause to select specific rows or create a *join* condition.
- Add an ORDER BY clause to change the order in which data is produced.
- Add an INTO TEMP clause to save the results as a table for further queries.

Two additional SELECT statement clauses, GROUP BY and HAVING, let you perform more complex data retrieval. They are introduced in Compose advanced SELECT statements on page 1619. Another clause, INTO, specifies the program or host variable to receive data from a SELECT statement in an application program. Complete syntax and rules for using the SELECT statement are in the *HCL® Informix® Guide to SQL: Syntax*.

Output from SELECT statements

Although the syntax remains the same across all HCL® Informix® products, the formatting and display of the resulting output depends on the application. The examples in this chapter and in Compose advanced SELECT statements on page 1619 display the SELECT statements and their output as they appear when you use the interactive Query-language option in .

Output from large object data types

When you issue a SELECT statement that includes a large object, displays the results as follows:

- For a TEXT column or CLOB column, the contents of the column are displayed.
- For a BYTE column, the words `<BYTE value>` are displayed instead of the actual value.
- For a BLOB column, the words `<SBlob data>` are displayed instead of the actual value.

Output from user-defined data types

uses special conventions to display output from columns that contain complex or opaque data types. For more information about these data types, refer to the *IBM® Informix® Database Design and Implementation Guide*.

Output in non-default code sets

You can issue a SELECT statement that queries NCHAR columns instead of CHAR columns or NVARCHAR columns instead of VARCHAR columns.

For more Global Language Support (GLS) information, see the *HCL® Informix® GLS User's Guide*. For additional information on using NCHAR and NVARCHAR data types with non-default code sets, see the *IBM® Informix® Database Design and Implementation Guide* and the *HCL® Informix® Guide to SQL: Reference*.

Some basic concepts

The SELECT statement, unlike INSERT, UPDATE, and DELETE statements, does not modify the data in a database. It simply queries the data. Whereas only one user at a time can modify data, multiple users can query or select the data concurrently. For more information about statements that modify data, see Modify data on page 1658. The syntax descriptions of the INSERT, UPDATE, and DELETE statements appear in the *HCL® Informix® Guide to SQL: Syntax*.

In a relational database, a *column* is a data element that contains a particular type of information that occurs in every row in the table. A *row* is a group of related items of information about a single entity across all columns in a database table.

You can select columns and rows from a database table; from a *system catalog table*, a special table that contains information on the database; or from a *view*, a virtual table created to contain a customized set of data. System catalog tables are described in the *HCL® Informix® Guide to SQL: Reference*. Views are discussed in the *IBM® Informix® Database Design and Implementation Guide*.

Privileges

Before you make a query against data, make sure you have the Connect privilege on the database and the Select privilege on the table. These privileges are normally granted to all users. Database access privileges are discussed in the *IBM® Informix® Database Design and Implementation Guide* and in the GRANT and REVOKE statements in the *HCL® Informix® Guide to SQL: Syntax*.

Relational operations

A *relational operation* involves manipulating one or more tables, or *relations*, to result in another table. The three kinds of relational operations are selection, projection, and join. This chapter includes examples of selection, projection, and simple joining.

Selection and projection

In relational terminology, *selection* is defined as taking the horizontal subset of rows of a single table that satisfies a particular condition. This kind of SELECT statement returns some of the rows and all the columns in a table. Selection is implemented through the WHERE clause of a SELECT statement, as the following figure shows.

Figure 25. Query

```
SELECT * FROM customer WHERE state = 'NJ';
```

The result contains the same number of columns as the **customer** table, but only a subset of its rows. In this example, displays the data from each column on a separate line.

Figure 26. Query result

```
customer_num  119
fname         Bob
lname         Shorter
company       The Triathletes Club
address1      2405 Kings Highway
address2
city          Cherry Hill
state         NJ
zipcode       08002
phone         609-663-6079

customer_num  122
fname         Cathy
lname         O'Brian
company       The Sporting Life
address1      543d Nassau
address2
city          Princeton
state         NJ
zipcode       08540
phone         609-342-0054
```

In relational terminology, *projection* is defined as taking a vertical subset from the columns of a single table that retains the unique rows. This kind of SELECT statement returns some of the columns and all the rows in a table.

Projection is implemented through the *projection list* in the Projection clause of a SELECT statement, as the following figure shows.

Figure 27. Query

```
SELECT city, state, zipcode FROM customer;
```

The result contains the same number of rows as the **customer** table, but it *projects* only a subset of the columns in the table. Because only a small amount of data is selected from each row, is able to display all of the data from the row on one line.

Figure 28. Query result

```
city            state zipcode

Sunnyvale       CA    94086
San Francisco   CA    94117
Palo Alto       CA    94303
Redwood City    CA    94026
Los Altos       CA    94022
Mountain View   CA    94063
Palo Alto       CA    94304
Redwood City    CA    94063
Sunnyvale       CA    94086
Redwood City    CA    94062
Sunnyvale       CA    94085

Oakland         CA    94609
Cherry Hill     NJ    08002
Phoenix         AZ    85016
Wilmington      DE    19898
Princeton       NJ    08540
Jacksonville    FL    32256
Bartlesville    OK    74006
```

The most common kind of SELECT statement uses both selection and projection. A query of this kind returns some of the rows and some of the columns in a table, as the following figure shows.

Figure 29. Query

```
SELECT UNIQUE city, state, zipcode
   FROM customer
   WHERE state = 'NJ';
```

Figure 30: Query result on page 1536 contains a subset of the rows and a subset of the columns in the **customer** table.

Figure 30. Query result

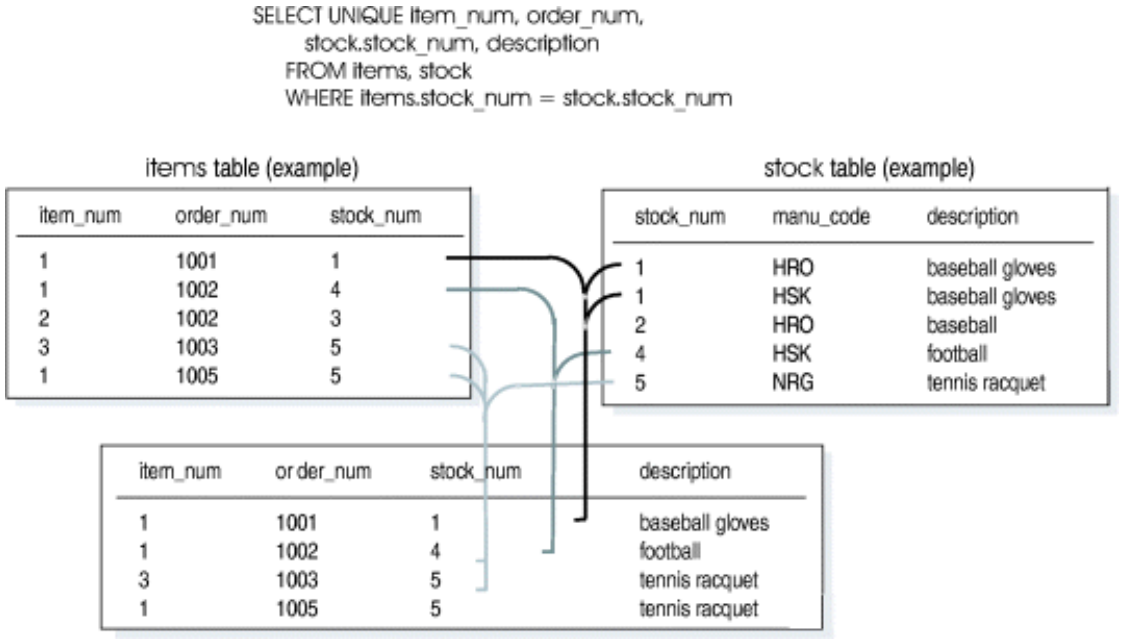
```
city            state zipcode

Cherry Hill     NJ    08002
Princeton       NJ    08540
```

Join

A join occurs when two or more tables are connected by one or more columns in common, which creates a new table of results. The following figure shows a query that uses a subset of the **items** and **stock** tables to illustrate the concept of a join.

Figure 31. A join between two tables



The following query joins the **customer** and **state** tables.

Figure 32. Query

```
SELECT UNIQUE city, state, zipcode, sname
   FROM customer, state
   WHERE customer.state = state.code;
```

The result consists of specified rows and columns from both the **customer** and **state** tables.

Figure 33. Query result

```
city           state zipcode sname

Bartlesville    OK    74006   Oklahoma
Blue Island     NY    60406   New York
Brighton        MA    02135   Massachusetts
Cherry Hill     NJ    08002   New Jersey
Denver          CO    80219   Colorado
Jacksonville    FL    32256   Florida
Los Altos       CA    94022   California
Menlo Park      CA    94025   California
Mountain View   CA    94040   California
Mountain View   CA    94063   California
Oakland         CA    94609   California
Palo Alto       CA    94303   California
Palo Alto       CA    94304   California
Phoenix         AZ    85008   Arizona
Phoenix         AZ    85016   Arizona
Princeton       NJ    08540   New Jersey
Redwood City    CA    94026   California
Redwood City    CA    94062   California
Redwood City    CA    94063   California
San Francisco   CA    94117   California
Sunnyvale       CA    94085   California
Sunnyvale       CA    94086   California
Wilmington      DE    19898   Delaware
```

Single-table SELECT statements

You can query a single table in a database in many ways. You can tailor a SELECT statement to perform the following actions:

- Retrieve all or specific columns
- Retrieve all or specific rows
- Perform computations or other functions on the retrieved data
- Order the data in various ways

The most basic SELECT statement contains only the two required clauses, the Projection clause and FROM.

The asterisk symbol (*)

The following query specifies all the columns in the **manufact** table in a projection list. An *explicit* projection list is a list of the column names or expressions that you want to project from a table.

Figure 34. Query

```
SELECT manu_code, manu_name, lead_time FROM manufact;
```

The following query uses the *wildcard* asterisk symbol (*) as shorthand in the projection list to represent the names of all the columns in the table. You can use the asterisk symbol (*) when you want all the columns in their defined order. An *implicit* select list uses the asterisk symbol.

Figure 35. Query

```
SELECT * FROM manufact;
```

Because the **manufact** table has only three columns, Figure 34: Query on page 1538 and Figure 35: Query on page 1539 are equivalent and display the same results; that is, a list of every column and row in the **manufact** table. The following figure shows the results.

Figure 36. Query result

```
manu_code manu_name       lead_time

 SMT       Smith              3
 ANZ       Anza               5
 NRG       Norge              7
 HSK       Husky              5
 HRO       Hero               4
 SHM       Shimara           30
 KAR       Karsten           21
 NKL       Nikolus            8
 PRC       ProCycle           9
```

Reorder the columns

The following query shows how you can change the order in which the columns are listed by changing their order in your projection list.

Figure 37. Query

```
SELECT manu_name, manu_code, lead_time FROM manufact;
```

The query result includes the same columns as the previous query result, but because the columns are specified in a different order, the display is also different.

Figure 38. Query result

```
manu_name       manu_code lead_time

 Smith           SMT          3
 Anza            ANZ          5
 Norge           NRG          7
 Husky           HSK          5
 Hero            HRO          4
 Shimara         SHM         30
 Karsten         KAR         21
 Nikolus         NKL          8
 ProCycle        PRC          9
```

The ORDER BY clause to sort the rows

The results from a query are not arranged in any particular order. For example, Figure 28: Query result on page 1536 and Figure 38: Query result on page 1539 appear to be in random order.

You can add an ORDER BY clause to your SELECT statement to direct the system to sort the data in a specific order. The ORDER BY clause is a list of column names from any remote or local table or view. Any expressions that are allowed in the

projection list are allowed in the ORDER BY list. If a column used in the ORDER BY list has a Select trigger on it, the trigger will not be activated.

The following query returns every row from the **manu_code**, **manu_name**, and **lead_time** columns in the **manufact** table, sorted according to **lead_time**.

Figure 39. Query

```
SELECT manu_code, manu_name, lead_time
   FROM manufact
   ORDER BY lead_time;
```

For , you do not need to include the columns that you want to use in the ORDER BY clause in the projection list. That is, you can sort the data according to a column that is not retrieved in the projection list. The following query returns every row from the **manu_code** and **manu_name** columns in the **manufact** table, sorted according to **lead_time**. The **lead_time** column is in the ORDER BY clause although it is not included in the projection list.

Figure 40. Query

```
SELECT manu_code, manu_name
   FROM manufact
   ORDER BY lead_time;
```

Ascending order

The retrieved data is sorted and displayed, by default, in *ascending* order. In the ASCII character set, ascending order is uppercase `A` to lowercase `z` for character data types, and lowest to highest value for numeric data types. DATE and DATETIME data is sorted from earliest to latest, and INTERVAL data is ordered from shortest to longest span of time.

Descending order

Descending order is the opposite of ascending order, from lowercase `z` to uppercase `A` for character types, and from highest to lowest for numeric data types. DATE and DATETIME data is sorted from latest to earliest, and INTERVAL data is ordered from longest to shortest span of time. The following query shows an example of descending order.

Figure 41. Query

```
SELECT * FROM manufact ORDER BY lead_time DESC;
```

The keyword DESC following a column name causes the retrieved data to be sorted in descending order, as the result shows.

Figure 42. Query result

```
manu_code manu_name       lead_time

 SHM       Shimara           30
 KAR       Karsten           21
 PRC       ProCycle           9
 NKL       Nikolus            8
 NRG       Norge              7
 HSK       Husky              5
 ANZ       Anza               5
 HRO       Hero               4
 SMT       Smith              3
```

You can specify any column of a built-in data type (except TEXT, BYTE, BLOB, or CLOB) in the ORDER BY clause, and the database server sorts the data based on the values in that column.

Sorting on multiple columns

You can also ORDER BY two or more columns, which creates a *nested sort*. The default is still ascending, and the column that is listed first in the ORDER BY clause takes precedence.

The following query and Figure 45: Query on page 1541 and the corresponding query results show nested sorts. To modify the order in which selected data is displayed, change the order of the two columns that are named in the ORDER BY clause.

Figure 43. Query

```
SELECT stock_num, manu_code, description, unit_price
   FROM stock
   ORDER BY manu_code, unit_price;
```

In the query result, the **manu_code** column data appears in alphabetical order and, within each set of rows with the same **manu_code** (for example, ANZ, HRO), the **unit_price** is listed in ascending order.

Figure 44. Query result

```
stock_num manu_code description     unit_price

        5 ANZ       tennis racquet      $19.80
        9 ANZ       volleyball net      $20.00
        6 ANZ       tennis ball         $48.00
      313 ANZ       swim cap            $60.00
      201 ANZ       golf shoes          $75.00
      310 ANZ       kick board          $84.00
      ⋮
      111 SHM       10-spd, assmbld    $499.99
      112 SHM       12-spd, assmbld    $549.00
      113 SHM       18-spd, assmbld    $685.90
        5 SMT       tennis racquet      $25.00
        6 SMT       tennis ball         $36.00
        1 SMT       baseball gloves    $450.00
```

The following query shows the reverse order of the columns in the ORDER BY clause.

Figure 45. Query

```
SELECT stock_num, manu_code, description, unit_price
   FROM stock
   ORDER BY unit_price, manu_code;
```

In the query result, the data appears in ascending order of **unit_price** and, where two or more rows have the same **unit_price** (for example, $20.00, $48.00, $312.00), the **manu_code** is in alphabetical order.

Figure 46. Query result

```
stock_num manu_code description     unit_price

      302 HRO       ice pack             $4.50
      302 KAR       ice pack             $5.00
        5 ANZ       tennis racquet      $19.80
        9 ANZ       volleyball net      $20.00
      103 PRC       frnt derailleur     $20.00
      ⋮
      108 SHM       crankset            $45.00
        6 ANZ       tennis ball         $48.00
      305 HRO       first-aid kit       $48.00
      303 PRC       socks               $48.00
      311 SHM       water gloves        $48.00
      ⋮
      113 SHM       18-spd, assmbld    $685.90
        1 HSK       baseball gloves    $800.00
        8 ANZ       volleyball         $840.00
        4 HSK       football           $960.00
```

The order of the columns in the ORDER BY clause is important, and so is the position of the DESC keyword. Although the statements in the following query contain the same components in the ORDER BY clause, each produces a different result (not shown).

Figure 47. Query

```
SELECT * FROM stock ORDER BY manu_code, unit_price DESC;

SELECT * FROM stock ORDER BY unit_price, manu_code DESC;

SELECT * FROM stock ORDER BY manu_code DESC, unit_price;

SELECT * FROM stock ORDER BY unit_price DESC, manu_code;
```

Select specific columns

The previous section shows how to select and order all data from a table. However, often all you want to see is the data in one or more specific columns. Again, the formula is to use the Projection and FROM clauses, specify the columns and table, and perhaps order the data in ascending or descending order with an ORDER BY clause.

If you want to find all the customer numbers in the **orders** table, use a statement such as the one in the following query.

Figure 48. Query

```
SELECT customer_num FROM orders;
```

The result shows how the statement simply selects all data in the **customer_num** column in the **orders** table and lists the customer numbers on all the orders, including duplicates.

Figure 49. Query result

```
customer_num

         104
         101
         104
         ⋮
         122
         123
         124
         126
         127
```

The output includes several duplicates because some customers have placed more than one order. Sometimes you want to see duplicate rows in a projection. At other times, you want to see only the distinct values, not how often each value appears.

To suppress duplicate rows, you can include the keyword DISTINCT or its synonym UNIQUE at the start of the select list, once in each level of a query, as the following query shows.

Figure 50. Query

```
SELECT DISTINCT customer_num FROM orders;

SELECT UNIQUE customer_num FROM orders;
```

To produce a more readable list, Figure 50: Query on page 1543 limits the display to show each customer number in the **orders** table only once, as the result shows.

Figure 51. Query result

```
customer_num

         101
         104
         106
         110
         111
         112
         115
         116
         117
         119
         120
         121
         122
         123
         124
         126
         127
```

Suppose you are handling a customer call, and you want to locate purchase order number DM354331. To list all the purchase order numbers in the **orders** table, use a statement such as the following query shows.

Figure 52. Query

```
SELECT po_num FROM orders;
```

The result shows how the statement retrieves data in the **po_num** column in the **orders** table.

Figure 53. Query result

```
po_num

B77836
9270
B77890
8006
2865
Q13557
278693
⋮
```

However, the list is not in a useful order. You can add an ORDER BY clause to sort the column data in ascending order and make it easier to find that particular **po_num**, as shown in the following query.

Figure 54. Query

```
SELECT po_num FROM orders ORDER BY po_num;
```

Figure 55. Query result

```
po_num

278693
278701
2865
429Q
4745
8006
8052
9270
B77836
B77890
⋮
```

To select multiple columns from a table, list them in the projection list in the Projection clause. The following query shows that the order in which the columns are selected is the order in which they are retrieved, from left to right.

Figure 56. Query

```
SELECT ship_date, order_date, customer_num,
       order_num, po_num
   FROM orders
   ORDER BY order_date, ship_date;
```

As Sorting on multiple columns on page 1541 shows, you can use the ORDER BY clause to sort the data in ascending or descending order and perform nested sorts. The result shows ascending order.

Figure 57. Query result

```
ship_date  order_date customer_num   order_num po_num

06/01/1998 05/20/1998          104        1001 B77836
05/26/1998 05/21/1998          101        1002 9270
05/23/1998 05/22/1998          104        1003 B77890
05/30/1998 05/22/1998          106        1004 8006
06/09/1998 05/24/1998          116        1005 2865
           05/30/1998          112        1006 Q13557
06/05/1998 05/31/1998          117        1007 278693
07/06/1998 06/07/1998          110        1008 LZ230
06/21/1998 06/14/1998          111        1009 4745
06/29/1998 06/17/1998          115        1010 429Q
06/29/1998 06/18/1998          117        1012 278701
07/03/1998 06/18/1998          104        1011 B77897
07/10/1998 06/22/1998          104        1013 B77930
07/03/1998 06/25/1998          106        1014 8052
07/16/1998 06/27/1998          110        1015 MA003
07/12/1998 06/29/1998          119        1016 PC6782
07/13/1998 07/09/1998          120        1017 DM354331
07/13/1998 07/10/1998          121        1018 S22942
07/16/1998 07/11/1998          122        1019 Z55709
07/16/1998 07/11/1998          123        1020 W2286
07/25/1998 07/23/1998          124        1021 C3288
07/30/1998 07/24/1998          126        1022 W9925
07/30/1998 07/24/1998          127        1023 KF2961
```

When you use SELECT and ORDER BY on several columns in a table, you might find it helpful to use integers to refer to the position of the columns in the ORDER BY clause. When an integer is an element in the ORDER BY list, the database server treats it as the position in the projection list. For example, using 3 in the ORDER BY list (ORDER BY 3) refers to the third item in the projection list. The statements in the following query retrieve and display the same data, as Figure 59: Query result on page 1546 shows.

Figure 58. Query

```
SELECT customer_num, order_num, po_num, order_date
   FROM orders
   ORDER BY 4, 1;

SELECT customer_num, order_num, po_num, order_date
   FROM orders
   ORDER BY order_date, customer_num;
```

Figure 59. Query result

```
customer_num   order_num po_num     order_date

         104        1001 B77836     05/20/1998
         101        1002 9270       05/21/1998
         104        1003 B77890     05/22/1998
         106        1004 8006       05/22/1998
         116        1005 2865       05/24/1998
         112        1006 Q13557     05/30/1998
         117        1007 278693     05/31/1998
         110        1008 LZ230      06/07/1998
         111        1009 4745       06/14/1998
         115        1010 429Q       06/17/1998
         104        1011 B77897     06/18/1998
         117        1012 278701     06/18/1998
         104        1013 B77930     06/22/1998
         106        1014 8052       06/25/1998
         110        1015 MA003      06/27/1998
         119        1016 PC6782     06/29/1998
         120        1017 DM354331   07/09/1998
         121        1018 S22942     07/10/1998
         122        1019 Z55709     07/11/1998
         123        1020 W2286      07/11/1998
         124        1021 C3288      07/23/1998
         126        1022 W9925      07/24/1998
         127        1023 KF2961     07/24/1998
```

You can include the DESC keyword in the ORDER BY clause when you assign integers to column names, as the following query shows.

Figure 60. Query

```
SELECT customer_num, order_num, po_num, order_date
   FROM orders
   ORDER BY 4 DESC, 1;
```

In this case, data is first sorted in descending order by **order_date** and in ascending order by **customer_num**.

Select substrings

To select part of the value of a character column, include a *substring* in the projection list. Suppose your marketing department is planning a mailing to your customers and wants their geographical distribution based on zip codes. You could write a query similar to the following.

Figure 61. Query

```
SELECT zipcode[1,3], customer_num
   FROM customer
   ORDER BY zipcode;
```

The query uses a substring to select the first three characters of the **zipcode** column (which identify the state) and the full **customer_num**, and lists them in ascending order by zip code, as the result shows.

Figure 62. Query result

```
zipcode customer_num

021             125
080             119
085             122
198             121
322             123
⋮
943             103
943             107
946             118
```

The WHERE clause

The set of rows that a SELECT statement returns is its *active set*. A *singleton* SELECT statement returns a single row. You can add a WHERE clause to a SELECT statement if you want to see only specific rows. For example, you use a WHERE clause to restrict the rows that the database server returns to only the orders that a particular customer placed or the calls that a particular customer service representative entered.

You can use the WHERE clause to set up a comparison condition or a join condition. This section demonstrates only the first use. Join conditions are described in a later section and in the next chapter.

Create a comparison condition

The WHERE clause of a SELECT statement specifies the rows that you want to see. A comparison condition employs specific *keywords* and *operators* to define the search criteria.

For example, you might use one of the keywords BETWEEN, IN, LIKE, or MATCHES to test for equality, or the keywords IS NULL to test for null values. You can combine the keyword NOT with any of these keywords to specify the opposite condition.

The following table lists the relational operators that you can use in a WHERE clause in place of a keyword to test for equality.

Operator

Operation

=

equals

!= or <>

does not equal

>

greater than

>=

greater than or equal to

<

less than

<=

less than or equal to

For CHAR expressions, greater than means *after* in ASCII collating order, where lowercase letters are after uppercase letters, and both are after numerals. See the ASCII Character Set chart in the *HCL® Informix® Guide to SQL: Syntax*. For DATE and DATETIME expressions, greater than means *later in time*, and for INTERVAL expressions, it means *of longer duration*.

You cannot use TEXT or BYTE columns to create a comparison condition, except when you use the IS NULL or IS NOT NULL keywords to test for NULL values.

You cannot specify BLOB or CLOB columns to create a comparison condition on , except when you use the IS NULL or IS NOT NULL keywords to test for NULL values.

You can use the preceding keywords and operators in a WHERE clause to create comparison-condition queries that perform the following actions:

- Include values
- Exclude values
- Find a range of values
- Find a subset of values
- Identify NULL values

To perform variable text searches using the following criteria, use the preceding keywords and operators in a WHERE clause to create comparison-condition queries:

- Exact-text comparison
- Single-character wildcards
- Restricted single-character wildcards
- Variable-length wildcards
- Subscripting

The following section contains examples that illustrate these types of queries.

Include rows

Use the equal sign (=) relational operator to include rows in a WHERE clause, as the following query shows.
Figure 63. Query

```
SELECT customer_num, call_code, call_dtime, res_dtime
   FROM cust_calls
   WHERE user_id = 'maryj';
```

The query returns the set of rows that is shown.

Figure 64. Query result

```
customer_num  call_code  call_dtime       res_dtime

         106  D          1998-06-12  08:20 1998-06-12 08:25
         121  O          1998-07-10 14:05 1998-07-10 14:06
         127  I          1998-07-31 14:30
```

Exclude rows

Use the relational operators != or <> to exclude rows in a WHERE clause.

The following query assumes that you are selecting from an ANSI-compliant database; the statements specify the owner or login name of the creator of the **customer** table. This qualifier is not required when the creator of the table is the current user, or when the database is not ANSI compliant. However, you can include the qualifier in either case. For a detailed discussion of owner naming, see the *HCL® Informix® Guide to SQL: Syntax*.

Figure 65. Query

```
SELECT customer_num, company, city, state
   FROM odin.customer
   WHERE state != 'CA';

SELECT customer_num, company, city, state
   FROM odin.customer
   WHERE state <> 'CA';
```

Both statements in the query exclude values by specifying that, in the **customer** table that the user **odin** owns, the value in the **state** column should not be equal to `CA`, as the result shows.

Figure 66. Query result

```
customer_num  company              city            state

         119  The Triathletes Club Cherry Hill     NJ
         120  Century Pro Shop     Phoenix         AZ
         121  City Sports          Wilmington      DE
         122  The Sporting Life    Princeton       NJ
         123  Bay Sports           Jacksonville    FL
         124  Putnum's Putters     Bartlesville    OK
         125  Total Fitness Sports Brighton        MA
         126  Neelie's Discount Sp Denver          CO
         127  Big Blue Bike Shop   Blue Island     NY
         128  Phoenix College      Phoenix         AZ
```

Specify a range of rows

The following query shows two ways to specify a range of rows in a WHERE clause.

Figure 67. Query

```
SELECT catalog_num, stock_num, manu_code, cat_advert
   FROM catalog
   WHERE catalog_num BETWEEN 10005 AND 10008;

SELECT catalog_num, stock_num, manu_code, cat_advert
   FROM catalog
   WHERE catalog_num >= 10005 AND catalog_num <= 10008;
```

Each statement in the query specifies a range for **catalog_num** from 10005 through 10008, inclusive. The first statement uses keywords, and the second statement uses relational operators to retrieve the rows, as the result shows.

Figure 68. Query result

```
catalog_num  10005
stock_num    3
manu_code    HSK
cat_advert   High-Technology Design Expands the Sweet Spot

catalog_num  10006
stock_num    3
manu_code    SHM
cat_advert   Durable Aluminum for High School and Collegiate Athletes

catalog_num  10007
stock_num    4
manu_code    HSK
cat_advert   Quality Pigskin with Joe Namath Signature

catalog_num  10008
stock_num    4
manu_code    HRO
cat_advert   Highest Quality Football for High School
              and Collegiate Competitions
```

Although the **catalog** table includes a column with the BYTE data type, that column is not included in this SELECT statement because the output would show only the words `<BYTE value>` by the column name. You can write an SQL API application to display TEXT and BYTE values.

Exclude a range of rows

The following query uses the keywords NOT BETWEEN to exclude rows that have the character range 94000 through 94999 in the **zipcode** column, as the result shows.

Figure 69. Query

```
SELECT fname, lname, city, state
   FROM customer
   WHERE zipcode NOT BETWEEN '94000' AND '94999'
   ORDER BY state;
```

Figure 70. Query result

```
fname           lname           city            state

Frank           Lessor          Phoenix         AZ
Fred            Jewell          Phoenix         AZ
Eileen          Neelie          Denver          CO
Jason           Wallack         Wilmington      DE
Marvin          Hanlon          Jacksonville    FL
James           Henry           Brighton        MA
Bob             Shorter         Cherry Hill     NJ
Cathy           O'Brian         Princeton       NJ
Kim             Satifer         Blue Island     NY
Chris           Putnum          Bartlesville    OK
```

Use a WHERE clause to find a subset of values

Like Exclude rows on page 1549, the following query assumes the use of an ANSI-compliant database. The owner qualifier is in quotation marks to preserve the case sensitivity of the literal string.

Figure 71. Query

```
SELECT lname, city, state, phone
   FROM 'Aleta'.customer
   WHERE state = 'AZ' OR state = 'NJ'
   ORDER BY lname;

SELECT lname, city, state, phone
   FROM 'Aleta'.customer
   WHERE state IN ('AZ', 'NJ')
   ORDER BY lname;
```

Each statement in the query retrieves rows that include the subset of `AZ` or `NJ` in the **state** column of the **Aleta**.**customer** table.

Figure 72. Query result

```
lname      city            state phone

Jewell     Phoenix         AZ    602-265-8754
Lessor     Phoenix         AZ    602-533-1817
O'Brian    Princeton       NJ    609-342-0054
Shorter    Cherry Hill     NJ    609-663-6079
```

You cannot test TEXT or BYTE columns with the IN keyword.

Also, when you use , you cannot test BLOB or CLOB columns with the IN keyword.

In the example of a query on an ANSI-compliant database, no quotation marks exist around the table owner name. Whereas the two statements in Figure 71: Query on page 1551 searched the **Aleta**.**customer** table, the following query searches the table ALETA.**customer**, which is a different table, because of the way ANSI-compliant databases look at owner names.

Figure 73. Query

```
SELECT lname, city, state, phone
   FROM Aleta.customer
   WHERE state NOT IN ('AZ', 'NJ')
   ORDER BY state;
```

The previous query adds the keywords NOT IN, so the subset changes to exclude the subsets `AZ` and `NJ` in the **state** column. The following figure shows the results in order of the **state** column.

Figure 74. Query result

```
lname        city            state phone

Pauli        Sunnyvale       CA    408-789-8075
Sadler       San Francisco   CA    415-822-1289
Currie       Palo Alto       CA    415-328-4543
Higgins      Redwood City    CA    415-368-1100
Vector       Los Altos       CA    415-776-3249
Watson       Mountain View   CA    415-389-8789
Ream         Palo Alto       CA    415-356-9876
Quinn        Redwood City    CA    415-544-8729
Miller       Sunnyvale       CA    408-723-8789
Jaeger       Redwood City    CA    415-743-3611
Keyes        Sunnyvale       CA    408-277-7245
Lawson       Los Altos       CA    415-887-7235
Beatty       Menlo Park      CA    415-356-9982
Albertson    Redwood City    CA    415-886-6677
Grant        Menlo Park      CA    415-356-1123
Parmelee     Mountain View   CA    415-534-8822
Sipes        Redwood City    CA    415-245-4578
Baxter       Oakland         CA    415-655-0011
Neelie       Denver          CO    303-936-7731
Wallack      Wilmington      DE    302-366-7511
Hanlon       Jacksonville    FL    904-823-4239
Henry        Brighton        MA    617-232-4159
Satifer      Blue Island     NY    312-944-5691
Putnum       Bartlesville    OK    918-355-2074
```

Identify NULL values

Use the IS NULL or IS NOT NULL option to check for NULL values. A NULL value represents either the absence of data or an unknown value. A NULL value is not the same as a zero or a blank.

The following query returns all rows that have a null **paid_date**, as the result shows.

Figure 75. Query

```
SELECT order_num, customer_num, po_num, ship_date
   FROM orders
   WHERE paid_date IS NULL
   ORDER BY customer_num;
```

Figure 76. Query result

```
order_num  customer_num  po_num   ship_date

     1004           106  8006     05/30/1998
     1006           112  Q13557
     1007           117  278693   06/05/1998
     1012           117  278701   06/29/1998
     1016           119  PC6782   07/12/1998
     1017           120  DM354331 07/13/1998
```

Form compound conditions

To connect two or more comparison conditions, or *Boolean expressions*, use the logical operators AND, OR, and NOT. A Boolean expression evaluates as `true` or `false` or, if NULL values are involved, as `unknown`.

In the following query, the operator AND combines two comparison expressions in the WHERE clause.

Figure 77. Query

```
SELECT order_num, customer_num, po_num, ship_date
   FROM orders
   WHERE paid_date IS NULL
      AND ship_date IS NOT NULL
   ORDER BY customer_num;
```

The query returns all rows that have NULL paid_date or a NOT NULL ship_date.

Figure 78. Query result

```
order_num customer_num po_num     ship_date

     1004          106 8006       05/30/1998
     1007          117 278693     06/05/1998
     1012          117 278701     06/29/1998
     1017          120 DM354331   07/13/1998
```

Exact-text comparisons

The following examples include a WHERE clause that searches for exact-text comparisons by using the keyword LIKE or MATCHES or the equal sign (=) relational operator. Unlike earlier examples, these examples illustrate how to query a table that is not in the current database. You can access a table that is not in the current database only if the database that contains the table has the same ANSI compliance status as the current database. If the current database is an ANSI-compliant database, the table you want to access must also reside in an ANSI-compliant database. If the current database is not an ANSI-compliant database, the table you want to access must also reside in a database that is not an ANSI-compliant database.

Although the database used previously in this chapter is the demonstration database, the FROM clause in the following examples specifies the **manatee** table, created by the owner **bubba**, which resides in an ANSI-compliant database named **syzygy**. For more information on how to access tables that are not in the current database, see the *HCL® Informix® Guide to SQL: Syntax*.

Each statement in the following query retrieves all the rows that have the single word *helmet* in the **description** column, as the result shows.

Figure 79. Query

```
SELECT stock_no, mfg_code, description, unit_price
   FROM syzygy:bubba.manatee
   WHERE description = 'helmet'
   ORDER BY mfg_code;

SELECT stock_no, mfg_code, description, unit_price
   FROM syzygy:bubba.manatee
   WHERE description LIKE 'helmet'
   ORDER BY mfg_code;

SELECT stock_no, mfg_code, description, unit_price
   FROM syzygy:bubba.manatee
   WHERE description MATCHES 'helmet'
   ORDER BY mfg_code;
```

The results might look like the following figure.

Figure 80. Query result

```
stock_no mfg_code  description     unit_price

    991 ABC       helmet             $222.00
    991 BKE       helmet             $269.00
    991 HSK       helmet             $311.00
    991 PRC       helmet             $234.00
    991 SPR       helmet             $245.00
```

Variable-text searches

You can use the keywords LIKE and MATCHES for variable-text queries that are based on substring searches of fields. Include the keyword NOT to indicate the opposite condition.

The keyword LIKE complies with the ISO/ANSI standard for SQL, whereas MATCHES is the HCL® Informix® extension.

Variable-text search strings can include the wildcard symbols that are listed with the keywords LIKE or MATCHES in the following table.

Keyword	Symbol	Explanation
LIKE	%	Evaluates to zero or more characters
LIKE	_	Evaluates to a single character
LIKE	\	Escapes special significance of next character
MATCHES	*	Evaluates to zero or more characters
MATCHES	?	Evaluates to a single character (except null)

Keyword	Symbol	Explanation
MATCHES	[ ]	Evaluates to a single character or range of values
MATCHES	\	Escapes special significance of next character

You cannot test BLOB, CLOB, TEXT, or BYTE columns with the LIKE or MATCHES operators.

A single-character wildcard

The statements in the following query illustrate the use of a single-character wildcard in a WHERE clause. Further, they demonstrate a query on a table that is not in the current database. The **stock** table is in the database **sloth**. Besides being outside the current demonstration database, **sloth** is on a separate database server called **meerkat**.

For more information, see Access and modify data in an external database on page 1696 and the *HCL® Informix® Guide to SQL: Syntax*.

Figure 81. Query

```
SELECT stock_num, manu_code, description, unit_price
   FROM sloth@meerkat:stock
   WHERE manu_code LIKE '_R_'
       AND unit_price >= 100
   ORDER BY description, unit_price;

SELECT stock_num, manu_code, description, unit_price
   FROM sloth@meerkat:stock
   WHERE manu_code MATCHES '?R?'
      AND unit_price >= 100
   ORDER BY description, unit_price;
```

Each statement in the query retrieves only those rows for which the middle letter of the **manu_code** is `R`, as the result shows. The comparison '`_R_`' (for LIKE) or '`?R?`' (for MATCHES) specifies, from left to right, the following items:

- Any single character
- The letter `R`
- Any single character

Figure 82. Query result

```
stock_num manu_code description     unit_price

      205 HRO       3 golf balls       $312.00
        2 HRO       baseball           $126.00
        1 HRO       baseball gloves    $250.00
        7 HRO       basketball         $600.00
      102 PRC       bicycle brakes     $480.00
      114 PRC       bicycle gloves     $120.00
        4 HRO       football           $480.00
      110 PRC       helmet             $236.00
      110 HRO       helmet             $260.00
      307 PRC       infant jogger      $250.00
      306 PRC       tandem adapter     $160.00
      308 PRC       twin jogger        $280.00
      304 HRO       watch              $280.00
```

WHERE clause to specify a range of initial characters

The following query selects only those rows where the **manu_code** begins with `A` through `H` and returns the rows that the result shows. The test '`[A-H]`' specifies any single letter from `A` through `H`, inclusive. No equivalent wildcard symbol exists for the LIKE keyword.

Figure 83. Query

```
SELECT stock_num, manu_code, description, unit_price
   FROM stock
   WHERE manu_code MATCHES '[A-H]*'
   ORDER BY description, manu_code;
```

Figure 84. Query result

```
stock_num manu_code description     unit_price

      205 ANZ       3 golf balls       $312.00
      205 HRO       3 golf balls       $312.00
        2 HRO       baseball           $126.00
        3 HSK       baseball bat       $240.00
        1 HRO       baseball gloves    $250.00
        1 HSK       baseball gloves    $800.00
        7 HRO       basketball         $600.00

      313 ANZ       swim cap            $60.00
        6 ANZ       tennis ball         $48.00
        5 ANZ       tennis racquet      $19.80
        8 ANZ       volleyball         $840.00
        9 ANZ       volleyball net      $20.00
      304 ANZ       watch              $170.00
```

WHERE clause with variable-length wildcard

The statements in the following query use a wildcard at the end of a string to retrieve all the rows where the **description** begins with the characters `bicycle`.

Figure 85. Query

```
SELECT stock_num, manu_code, description, unit_price
   FROM stock
   WHERE description LIKE 'bicycle%'
   ORDER BY description, manu_code;

SELECT stock_num, manu_code, description, unit_price
   FROM stock
   WHERE description MATCHES 'bicycle*'
   ORDER BY description, manu_code;
```

Either statement returns the following rows.

Figure 86. Query result

```
stock_num manu_code description     unit_price

      102 PRC       bicycle brakes     $480.00
      102 SHM       bicycle brakes     $220.00
      114 PRC       bicycle gloves     $120.00
      107 PRC       bicycle saddle      $70.00
      106 PRC       bicycle stem        $23.00
      101 PRC       bicycle tires       $88.00
      101 SHM       bicycle tires       $68.00
      105 PRC       bicycle wheels      $53.00
      105 SHM       bicycle wheels      $80.00
```

The comparison '`bicycle%`' or '`bicycle*`' specifies the characters `bicycle` followed by any sequence of zero or more characters. It matches `bicycle stem` with `stem` matched by the wildcard. It matches to the characters `bicycle` alone, if a row exists with that description.

The following query narrows the search by adding another comparison condition that excludes a **manu_code** of `PRC`.

Figure 87. Query

```
SELECT stock_num, manu_code, description, unit_price
   FROM stock
   WHERE description LIKE 'bicycle%'
      AND manu_code NOT LIKE 'PRC'
   ORDER BY description, manu_code;
```

The statement retrieves only the following rows.

Figure 88. Query result

```
stock_num manu_code description     unit_price

      102 SHM       bicycle brakes     $220.00
      101 SHM       bicycle tires       $68.00
      105 SHM       bicycle wheels      $80.00
```

When you select from a large table and use an initial wildcard in the comparison string (such as '`%cycle`'), the query often takes longer to execute. Because indexes cannot be used, every row is searched.

Protect special characters

The following query uses the keyword ESCAPE with LIKE or MATCHES so you can protect a special character from misinterpretation as a wildcard symbol.

Figure 89. Query

```
SELECT * FROM cust_calls
   WHERE res_descr LIKE '%!%%' ESCAPE '!';
```

The ESCAPE keyword designates an escape character (`!` in this example) that protects the next character so that it is interpreted as data and not as a wildcard. In the example, the escape character causes the middle percent sign (%) to be treated as data. By using the ESCAPE keyword, you can search for occurrences of a percent sign (%) in the **res_descr** column by using the LIKE wildcard percent sign (%). The query retrieves the following row.

Figure 90. Query result

```
customer_num   116
call_dtime     1997-12-21 11:24
user_id        mannyn
call_code      I
call_descr     Second complaint from this customer!
               Received two cases righthanded outfielder
               glove (1 HRO) instead of one case lefties.
res_dtime      1997-12-27 08:19
res_descr      Memo to shipping (Ava Brown) to send case
               of lefthanded gloves, pick up wrong case;
               memo to billing requesting 5% discount to
               placate customer due to second offense
               and lateness of resolution because of
               holiday.
```

Subscripting in a WHERE clause

You can use *subscripting* in the WHERE clause of a SELECT statement to specify a range of characters or numbers in a column, as the following query shows.

Figure 91. Query

```
SELECT catalog_num, stock_num, manu_code, cat_advert,
         cat_descr
   FROM catalog
   WHERE cat_advert[1,4] = 'High';
```

The subscript `[1,4]` causes the query to retrieve all rows in which the first four letters of the **cat_advert** column are `High`, as result shows.

Figure 92. Query result

```
catalog_num  10004
stock_num    2
manu_code    HRO
cat_advert   Highest Quality Ball Available, from Hand-Sti
             tching to the Robinson Signature
cat_descr
Jackie Robinson signature ball. Highest professional quality,
used by National League.

catalog_num  10005
stock_num    3
manu_code    HSK
cat_advert   High-Technology Design Expands the Sweet Spot
cat_descr
Pro-style wood. Available in sizes: 31, 32, 33, 34, 35.
⋮
catalog_num  10045
stock_num    204
manu_code    KAR
cat_advert   High-Quality Beginning Set of Irons. Appropriate
             for High School Competitions
cat_descr
Ideally balanced for optimum control. Nylon covered shaft.

catalog_num  10068
stock_num    310
manu_code    ANZ
cat_advert   High-Quality Kickboard
cat_descr
White. Standard size.
```

FIRST clause to select specific rows

You can include a FIRST *max* specification in the Projection clause of a SELECT statement, where *max* has an integer value, to instruct the query to return no more than the first *max* rows that match the conditions of the SELECT statement. You can also use the keyword LIMIT as a synonym for FIRST in this context (and only in this context). The rows that a SELECT statement with a FIRST clause returns might depend on whether the statement also includes an ORDER BY clause.

The keyword SKIP, followed by an unsigned integer, can precede the FIRST or LIMIT keyword in the Projection clause. The SKIP *offset* clause instructs the database server to exclude the first *offset* qualifying rows from the result set of the query before returning the number of rows that the FIRST clause specifies. In SPL routines, the parameter of SKIP, FIRST, or LIMIT can be a literal integer or a local SPL variable. If the Projection clause includes SKIP *offset* but no FIRST or LIMIT specification, then the query returns all of the qualifying rows except for the first *offset* rows.

The Projection clause cannot include the SKIP, FIRST, or LIMIT keywords in these contexts:

- when the SELECT statement is part of a view definition
- in a subquery, except in the FROM clause of the outer query
- in a cross-server distributed query in which a participating database server does not support the SKIP, FIRST, or LIMIT keywords.

For information about restrictions on use of the FIRST clause, see the description of the Projection clause of the SELECT statement in the *HCL® Informix® Guide to SQL: Syntax*.

FIRST clause without an ORDER BY clause

If you do not include an ORDER BY clause in a SELECT statement with a FIRST clause, any rows that match the conditions of the SELECT statement might be returned. In other words, the database server determines which of the qualifying rows to return, and the query result can vary depending on the query plan that the optimizer chooses.

The following query uses the FIRST clause to return the first five rows from the **state** table.

Figure 93. Query

```
SELECT FIRST 5 * FROM state;
```

Figure 94. Query result

```
code sname

AK   Alaska
HI   Hawaii
CA   California
OR   Oregon
WA   Washington
```

You can use a FIRST clause when you simply want to know the names of all the columns and the type of data that a table contains, or to test a query that otherwise would return many rows. The following query shows how to use the FIRST clause to return column values for the first row of a table.

Figure 95. Query

```
SELECT FIRST 1 * FROM orders;
```

Figure 96. Query result

```
order_num      1001
order_date     05/20/1998
customer_num   104
ship_instruct  express
backlog        n
po_num         B77836
ship_date      06/01/1998
ship_weight    20.40
ship_charge    $10.00
paid_date      07/22/1998
```

FIRST clause with an ORDER BY clause

You can include an ORDER BY clause in a SELECT statement with a FIRST clause to return rows that contain the highest or lowest values for a specified column. The following query shows a query that includes an ORDER BY clause to return (by alphabetical order) the first five states contained in the **state** table. The query, which is the same as Figure 93: Query on page 1560 except for the ORDER BY clause, returns a different set of rows than Figure 93: Query on page 1560.

Figure 97. Query

```
SELECT FIRST 5 * FROM state ORDER BY sname;
```

Figure 98. Query result

```
code sname

AL   Alabama
AK   Alaska
AZ   Arizona
AR   Arkansas
CA   California
```

The following query shows how to use a FIRST clause in a query with an ORDER BY clause to find the 10 most expensive items listed in the **stock** table.

Figure 99. Query

```
SELECT FIRST 10 description, unit_price
   FROM stock ORDER BY unit_price DESC;
```

Figure 100. Query result

```
description     unit_price

football           $960.00
volleyball         $840.00
baseball gloves    $800.00
18-spd, assmbld    $685.90
irons/wedge        $670.00
basketball         $600.00
12-spd, assmbld    $549.00
10-spd, assmbld    $499.99
football           $480.00
bicycle brakes     $480.00
```

Applications can use the SKIP and FIRST keywords of the Projection clause, in conjunction with the ORDER BY clause, to perform successive queries that incrementally retrieve all of the qualifying rows in subsets of some fixed size (for example, the maximum number of rows that are visible without scrolling a screen display). You can accomplish this by incrementing the *offset* parameter of the SKIP clause by the *max* parameter of the FIRST clause after each query. By imposing a unique order on the qualifying rows, the ORDER BY clause ensures that each query returns a disjunct subset of the qualifying rows.

The following query shows a query that includes SKIP, FIRST, and ORDER BY specifications to return (by alphabetical order) the sixth through tenth states in the **state** table, but not the first five states. This query resembles Figure 93: Query on page 1560, except that the SKIP 5 specification instructs the database server to returns a different set of rows than Figure 93: Query on page 1560.

Figure 101. Query

```
SELECT SKIP 5 FIRST 5 * FROM state ORDER BY sname;
```

Figure 102. Query result

```
code sname

CO   Colorado
CT   Connecticut
DE   Delaware
FL   Florida
GA   Georgia
```

If you use the SKIP, FIRST, or LIMIT keywords, you must take care to specify parameters that correspond to the design goals of your application. If the *offset* parameter of skip is larger than the number of qualifying rows, then any FIRST or LIMIT specification has no effect, and the query returns nothing.

Expressions and derived values

You are not limited to selecting columns by name. You can list an *expression* in the Projection clause of a SELECT statement to perform computations on column data and to display information *derived* from the contents of one or more columns.

An expression consists of a column name, a constant, a quoted string, a keyword, or any combination of these items connected by operators. It can also include host variables (program data) when the SELECT statement is embedded in a program.

Arithmetic expressions

An arithmetic expression contains at least one of the arithmetic operators listed in the following table and produces a number.

Operator	Operation
+	addition
-	subtraction
*	multiplication
/	division

You cannot use TEXT or BYTE columns in arithmetic expressions.

With , you cannot specify BLOB or CLOB in arithmetic expressions.

Arithmetic operations enable you to see the results of proposed computations without actually altering the data in the database. You can add an INTO TEMP clause to save the altered data in a temporary table for further reference,

computations, or impromptu reports. The following query calculates a 7 percent sales tax on the **unit_price** column when the **unit_price** is $400 or more (but does not update it in the database).

Figure 103. Query

```
SELECT stock_num, description, unit_price, unit_price * 1.07
   FROM stock
   WHERE unit_price >= 400;
```

The result appears in the **expression** column.

Figure 104. Query result

```
stock_num description     unit_price     (expression)

        1 baseball gloves    $800.00          $856.00
        1 baseball gloves    $450.00          $481.50
        4 football           $960.00         $1027.20
        4 football           $480.00          $513.60
        7 basketball         $600.00          $642.00
        8 volleyball         $840.00          $898.80
      102 bicycle brakes     $480.00          $513.60
      111 10-spd, assmbld    $499.99          $534.99
      112 12-spd, assmbld    $549.00          $587.43
      113 18-spd, assmbld    $685.90          $733.91
      203 irons/wedge        $670.00          $716.90
```

The following query calculates a surcharge of $6.50 on orders when the quantity ordered is less than 5.

Figure 105. Query

```
SELECT item_num, order_num, quantity,
       total_price, total_price + 6.50
   FROM items
   WHERE quantity < 5;
```

The result appears in the **expression** column.

Figure 106. Query result

```
item_num   order_num quantity total_price  (expression)

       1        1001        1     $250.00      $256.50
       1        1002        1     $960.00      $966.50
       2        1002        1     $240.00      $246.50
       1        1003        1      $20.00       $26.50
       2        1003        1     $840.00      $846.50
       1        1004        1     $250.00      $256.50
       2        1004        1     $126.00      $132.50
       3        1004        1     $240.00      $246.50
       4        1004        1     $800.00      $806.50

       1        1023        2      $40.00       $46.50
       2        1023        2     $116.00      $122.50
       3        1023        1      $80.00       $86.50
       4        1023        1     $228.00      $234.50
       5        1023        1     $170.00      $176.50
       6        1023        1     $190.00      $196.50
```

The following query calculates and displays in the **expression** column the interval between when the customer call was received (**call_dtime**) and when the call was resolved (**res_dtime**), in days, hours, and minutes.

Figure 107. Query

```
SELECT customer_num, call_code, call_dtime,
       res_dtime - call_dtime
   FROM cust_calls
   ORDER BY customer_num;
```

Figure 108. Query result

```
customer_num call_code call_dtime             (expression)

         106 D         1998-06-12 08:20         0 00:05
         110 L         1998-07-07 10:24         0 00:06
         116 I         1997-11-28 13:34         0 03:13
         116 I         1997-12-21 11:24         5 20:55
         119 B         1998-07-01 15:00         0 17:21
         121 O         1998-07-10 14:05         0 00:01
         127 I         1998-07-31 14:30
```

Display labels

You can assign a *display label* to a computed or derived data column to replace the default column header **expression**. In Figure 103: Query on page 1563, Figure 105: Query on page 1563, and Figure 109: Query on page 1564, the derived data appears in the **expression** column. The following query also presents derived values, but the column that displays the derived values has the descriptive header **taxed**.

Figure 109. Query

```
SELECT stock_num, description, unit_price,
       unit_price * 1.07 taxed
   FROM stock
   WHERE unit_price >= 400;
```

The result shows that the label **taxed** is assigned to the expression in the projection list that displays the results of the operation `unit_price * 1.07`.

Figure 110. Query result

```
stock_num description     unit_price            taxed

        1 baseball gloves    $800.00          $856.00
        1 baseball gloves    $450.00          $481.50
        4 football           $960.00         $1027.20
        4 football           $480.00          $513.60
        7 basketball         $600.00          $642.00
        8 volleyball         $840.00          $898.80
      102 bicycle brakes     $480.00          $513.60
      111 10-spd, assmbld    $499.99          $534.99
      112 12-spd, assmbld    $549.00          $587.43
      113 18-spd, assmbld    $685.90          $733.91
      203 irons/wedge        $670.00          $716.90
```

In the following query, the label **surcharge** is defined for the column that displays the results of the operation `total_price + 6.50`.

Figure 111. Query

```
SELECT item_num, order_num, quantity,
       total_price, total_price + 6.50 surcharge
   FROM items
   WHERE quantity < 5;
```

The **surcharge** column is labeled in the output.

Figure 112. Query result

```
item_num   order_num quantity total_price    surcharge

       1        1001        1     $250.00       $256.50
       1        1002        1     $960.00       $966.50
       2        1002        1     $240.00       $246.50
       1        1003        1      $20.00        $26.50
       2        1003        1     $840.00       $846.50

       1        1023        2      $40.00        $46.50
       2        1023        2     $116.00       $122.50
       3        1023        1      $80.00        $86.50
       4        1023        1     $228.00       $234.50
       5        1023        1     $170.00       $176.50
       6        1023        1     $190.00       $196.50
```

The following query assigns the label **span** to the column that displays the results of subtracting the DATETIME column **call_dtime** from the DATETIME column **res_dtime**.

Figure 113. Query

```
SELECT customer_num, call_code, call_dtime,
       res_dtime - call_dtime span
   FROM cust_calls
   ORDER BY customer_num;
```

The **span** column is labeled in the output.

Figure 114. Query result

```
customer_num call_code call_dtime              span

         106 D         1998-06-12 08:20         0 00:05
         110 L         1998-07-07 10:24         0 00:06
         116 I         1997-11-28 13:34         0 03:13
         116 I         1997-12-21 11:24         5 20:55
         119 B         1998-07-01 15:00         0 17:21
         121 O         1998-07-10 14:05         0 00:01
         127 I         1998-07-31 14:30
```

CASE expressions

A CASE expression is a conditional expression, which is similar to the concept of the CASE statement in programming languages. You can use a CASE expression when you want to change the way data is represented. The CASE expression

allows a statement to return one of several possible results, depending on which of several condition tests evaluates to TRUE.

TEXT or BYTE values are not allowed in a CASE expression.

Consider a column that represents marital status numerically as `1,2,3,4` with the corresponding values meaning single, married, divorced, widowed. In some cases, you might prefer to store the short values (`1,2,3,4`) for database efficiency, but employees in human resources might prefer the more descriptive values (single, married, divorced, widowed). The CASE expression makes such conversions between different sets of values easy.

In , the CASE expression also supports extended data types and cast expressions.

The following example shows a CASE expression with multiple WHEN clauses that returns more descriptive values for the **manu_code** column of the **stock** table. If none of the WHEN conditions is true, NULL is the default result. (You can omit the ELSE NULL clause.)

```
SELECT
   CASE
      WHEN manu_code = "HRO" THEN "Hero"
      WHEN manu_code = "SHM" THEN "Shimara"
      WHEN manu_code = "PRC" THEN "ProCycle"
      WHEN manu_code = "ANZ" THEN "Anza"
      ELSE NULL
   END
   FROM stock;
```

You must include at least one WHEN clause within the CASE expression; subsequent WHEN clauses and the ELSE clause are optional. If no WHEN condition evaluates to true, the resulting value is NULL. You can use the IS NULL expression to handle NULL results. For information on handling NULL values, see the *HCL® Informix® Guide to SQL: Syntax*.

The following query shows a simple CASE expression that returns a character string value to flag any orders from the **orders** table that have not been shipped to the customer.
Figure 115. Query

```
SELECT order_num, order_date,
      CASE
         WHEN ship_date IS NULL
         THEN "order not shipped"
      END
   FROM orders;
```

Figure 116. Query result

```
order_num order_date (expression)

       1001 05/20/1998
       1002 05/21/1998
       1003 05/22/1998
       1004 05/22/1998
       1005 05/24/1998
       1006 05/30/1998 order not shipped
       1007 05/31/1998

       1019 07/11/1998
       1020 07/11/1998
       1021 07/23/1998
       1022 07/24/1998
       1023 07/24/1998
```

For information about how to use the CASE expression to update a column, see CASE expression to update a column on page 1677.

Sorting on derived columns

When you want to use ORDER BY on an expression, you can use either the display label assigned to the expression or an integer, as Figure 117: Query on page 1567 and Figure 119: Query on page 1568 show.

Figure 117. Query

```
SELECT customer_num, call_code, call_dtime,
       res_dtime - call_dtime span
   FROM cust_calls
   ORDER BY span;
```

The query retrieves the same data from the **cust_calls** table as Figure 113: Query on page 1565. In the query, the ORDER BY clause causes the data to be displayed in ascending order of the derived values in the **span** column, as the result shows.

Figure 118. Query result

```
customer_num call_code call_dtime               span

         127 I         1998-07-31 14:30
         121 O         1998-07-10 14:05         0 00:01
         106 D         1998-06-12 08:20         0 00:05
         110 L         1998-07-07 10:24         0 00:06
         116 I         1997-11-28 13:34         0 03:13
         119 B         1998-07-01 15:00         0 17:21
         116 I         1997-12-21 11:24         5 20:55
```

The following query uses an integer to represent the result of the operation `res_dtime - call_dtime` and retrieves the same rows that appear in the above result.

Figure 119. Query

```
SELECT customer_num, call_code, call_dtime,
       res_dtime - call_dtime span
   FROM cust_calls
   ORDER BY 4;
```

Rowid values in SELECT statements

The database server assigns a unique *rowid* to rows in nonfragmented tables. The rowid is, in effect, a hidden column in every table. The sequential values of rowid have no special significance and can vary depending on the location of the physical data in the chunk. You can use a rowid to locate the internal record number that is associated with a row in a table. Rows in fragmented tables do not automatically contain the rowid column.

It is recommended that you use primary keys as a method of access in your applications rather than rowids. Because primary keys are defined in the ANSI specification of SQL, using them to access data makes your applications more portable. In addition, the database server requires less time to access data in a fragmented table when it uses a primary key than it requires to access the same data when it uses rowid.

For more information about rowids, see the *IBM® Informix® Database Design and Implementation Guide* and your *HCL® Informix® Administrator's Guide*.

The following query uses the rowid and the wildcard asterisk symbol (*) in the Projection clause to retrieve each row in the **manufact** table and its corresponding rowid.

Figure 120. Query

```
SELECT rowid, * FROM manufact;
```

Figure 121. Query result

```
rowid manu_code manu_name       lead_time

  257 SMT       Smith               3
  258 ANZ       Anza                5
  259 NRG       Norge               7
  260 HSK       Husky               5
  261 HRO       Hero                4
  262 SHM       Shimara            30
  263 KAR       Karsten            21
  264 NKL       Nikolus             8
  265 PRC       ProCycle            9
```

Never store a rowid in a permanent table or attempt to use it as a foreign key. If a table is dropped and then reloaded from external data, all the rowids will be different.

Multiple-table SELECT statements

To select data from two or more tables, specify the table names in the FROM clause. Add a WHERE clause to create a join condition between at least one related column in each table. This WHERE clause creates a temporary composite table in which each pair of rows that satisfies the join condition is linked to form a single row.

A *simple join* combines information from two or more tables based on the relationship between one column in each table. A *composite join* is a join between two or more tables based on the relationship between two or more columns in each table.

To create a join, you must specify a relationship, called a *join condition*, between at least one column from each table. Because the columns are being compared, they must have compatible data types. When you join large tables, performance improves when you index the columns in the join condition.

Data types are described in the *HCL® Informix® Guide to SQL: Reference* and the *IBM® Informix® Database Design and Implementation Guide*. Indexing is discussed in detail in the *HCL® Informix® Administrator's Guide*.

Create a Cartesian product

When you perform a multiple-table query that does not explicitly state a join condition among the tables, you create a *Cartesian product*. A Cartesian product consists of every possible combination of rows from the tables. This result is usually large and unwieldy.

The following query selects from two tables and produces a Cartesian product.
Figure 122. Query

```
SELECT * FROM customer, state;
```

Although only 52 rows exist in the **state** table and 28 rows in the **customer** table, the effect of the query is to multiply the rows of one table by the rows of the other and retrieve an impractical 1,456 rows, as the result shows.

Figure 123. Query result

```
customer_num  101
fname         Ludwig
lname         Pauli
company       All Sports Supplies
address1      213 Erstwild Court
address2
city          Sunnyvale
state         CA
zipcode       94086
phone         408-789-8075
code          AK
sname         Alaska

customer_num  101
fname         Ludwig
lname         Pauli
company       All Sports Supplies
address1      213 Erstwild Court
address2
city          Sunnyvale
state         CA
zipcode       94086
phone         408-789-8075
code          HI
sname         Hawaii

customer_num  101
fname         Ludwig
lname         Pauli
company       All Sports Supplies
address1      213 Erstwild Court
address2
city          Sunnyvale
state         CA
zipcode       94086
phone         408-789-8075
code          CA
sname         California
⋮
```

In addition, some of the data that is displayed in the concatenated rows is contradictory. For example, although the **city** and **state** from the **customer** table indicate an address in California, the **code** and **sname** from the **state** table might be for a different state.

Create a join

Conceptually, the first stage of any join is the creation of a Cartesian product. To refine or constrain this Cartesian product and eliminate meaningless combinations of rows of data, include a WHERE clause with a valid join condition in your SELECT statement.

This section illustrates *cross joins*, *equi-joins*, *natural joins*, and *multiple-table joins*. Additional complex forms, such as *self-joins* and *outer joins*, are discussed in Compose advanced SELECT statements on page 1619.

Cross join

A *cross join* combines all rows in all tables selected and creates a Cartesian product. The results of a cross join can be very large and difficult to manage.

The following query uses ANSI join syntax to create a cross join.
Figure 124. Query

```
SELECT * FROM customer CROSS JOIN  state;
```

The results of the query are identical to the results of Figure 122: Query on page 1569. In addition, you can filter a cross join by specifying a WHERE clause.

For more information about Cartesian products, see Create a Cartesian product on page 1569. For more information about ANSI syntax, see ANSI join syntax on page 1629.

Equi-join

An *equi-join* is a join based on equality or matching column values. This equality is indicated with an equal sign (=) as the comparison operator in the WHERE clause, as the following query shows.
Figure 125. Query

```
SELECT * FROM manufact, stock
   WHERE manufact.manu_code = stock.manu_code;
```

The query joins the **manufact** and **stock** tables on the **manu_code** column. It retrieves only those rows for which the values of the two columns are equal, some of which the result shows.

Figure 126. Query result

```
manu_code    SMT
manu_name    Smith
lead_time       3
stock_num    1
manu_code    SMT
description  baseball gloves
unit_price   $450.00
unit         case
unit_descr   10 gloves/case

manu_code    SMT
manu_name    Smith
lead_time       3
stock_num    5
manu_code    SMT
description  tennis racquet
unit_price   $25.00
unit         each
unit_descr   each

manu_code    SMT
manu_name    Smith
lead_time       3
stock_num    6
manu_code    SMT
description  tennis ball
unit_price   $36.00
unit         case
unit_descr   24 cans/case

manu_code    ANZ
manu_name    Anza
lead_time       5
stock_num    5
manu_code    ANZ
description  tennis racquet
unit_price   $19.80
unit         each
unit_descr   each
⋮
```

In this equi-join, the result includes the **manu_code** column from both the **manufact** and **stock** tables because the select list requested every column.

You can also create an equi-join with additional constraints, where the comparison condition is based on the inequality of values in the joined columns. These joins use a relational operator in addition to the equal sign (=) in the comparison condition that is specified in the WHERE clause.

To join tables that contain columns with the same name, qualify each column name with the name of its table and a period symbol (.), as the following query shows.

Figure 127. Query

```
SELECT order_num, order_date, ship_date, cust_calls.*
   FROM orders, cust_calls
   WHERE call_dtime >= ship_date
      AND cust_calls.customer_num = orders.customer_num
   ORDER BY orders.customer_num;
```

The query joins the **customer_num** column and then selects only those rows where the **call_dtime** in the **cust_calls** table is greater than or equal to the **ship_date** in the **orders** table. The result shows the combined rows that it returns.

Figure 128. Query result

```
order_num      1004
order_date     05/22/1998
ship_date      05/30/1998
customer_num   106
call_dtime     1998-06-12 08:20
user_id        maryj
call_code      D
call_descr     Order received okay, but two of the cans of
                       ANZ tennis balls within the case were empty
res_dtime      1998-06-12 08:25
res_descr      Authorized credit for two cans to customer,
               issued apology. Called ANZ buyer to report
               the qa problem.

order_num      1008
order_date     06/07/1998
ship_date      07/06/1998
customer_num   110
call_dtime     1998-07-07 10:24
user_id        richc
call_code      L
call_descr     Order placed one month ago (6/7) not received.
res_dtime      1998-07-07 10:30
res_descr      Checked with shipping (Ed Smith). Order out
               yesterday-was waiting for goods from ANZ.
               Next time will call with delay if necessary.

order_num      1023
order_date     07/24/1998
ship_date      07/30/1998
customer_num   127
call_dtime     1998-07-31 14:30
user_id        maryj
call_code      I
call_descr     Received Hero watches (item # 304) instead
                of ANZ watches
res_dtime
res_descr      Sent memo to shipping to send ANZ item 304
               to customer and pickup HRO watches. Should
               be done tomorrow, 8/1
```

Natural join

A *natural join* is a type of equi-join and is structured so that the join column does not display data redundantly, as the following query shows.

Figure 129. Query

```
SELECT manu_name, lead_time, stock.*
   FROM manufact, stock
   WHERE manufact.manu_code = stock.manu_code;
```

Like the example for equi-join, the query joins the **manufact** and **stock** tables on the **manu_code** column. Because the Projection list is more closely defined, the **manu_code** is listed only once for each row retrieved, as the result shows.

Figure 130. Query result

```
manu_name    Smith
lead_time       3
stock_num    1
manu_code    SMT
description  baseball gloves
unit_price   $450.00
unit         case
unit_descr   10 gloves/case

manu_name    Smith
lead_time       3
stock_num    5
manu_code    SMT
description  tennis racquet
unit_price   $25.00
unit         each
unit_descr   each

manu_name    Smith
lead_time       3
stock_num    6
manu_code    SMT
description  tennis ball
unit_price   $36.00
unit         case
unit_descr   24 cans/case

manu_name    Anza
lead_time       5
stock_num    5
manu_code    ANZ
description  tennis racquet
unit_price   $19.80
unit         each
unit_descr   each
```

All joins are *associative*; that is, the order of the joining terms in the WHERE clause does not affect the meaning of the join.

Both statements in the following query create the same natural join.

Figure 131. Query

```
SELECT catalog.*, description, unit_price, unit, unit_descr
   FROM catalog, stock
   WHERE catalog.stock_num = stock.stock_num
      AND catalog.manu_code = stock.manu_code
      AND catalog_num = 10017;

SELECT catalog.*, description, unit_price, unit, unit_descr
   FROM catalog, stock
   WHERE catalog_num = 10017
      AND catalog.manu_code = stock.manu_code
      AND catalog.stock_num = stock.stock_num;
```

Each statement retrieves the following row.

Figure 132. Query result

```
catalog_num  10017
stock_num    101
manu_code    PRC
cat_descr
Reinforced, hand-finished tubular. Polyurethane belted.
Effective against punctures. Mixed tread for super wear
and road grip.
cat_picture  <BYTE value>

cat_advert   Ultimate in Puncture Protection, Tires
             Designed for In-City Riding
description  bicycle tires
unit_price   $88.00
unit         box
unit_descr   4/box
```

Figure 131: Query on page 1575 includes a TEXT column, **cat_descr**; a BYTE column, **cat_picture**; and a VARCHAR column, **cat_advert**.

Multiple-table join

A *multiple-table join* connects more than two tables on one or more associated columns; it can be an equi-join or a natural join.

The following query creates an equi-join on the **catalog**, **stock**, and **manufact** tables.

Figure 133. Query

```
SELECT * FROM catalog, stock, manufact
   WHERE catalog.stock_num = stock.stock_num
      AND stock.manu_code = manufact.manu_code
      AND catalog_num = 10025;
```

The query retrieves the following rows.

Figure 134. Query result

```
catalog_num  10025
stock_num    106
manu_code    PRC
cat_descr
Hard anodized alloy with pearl finish; 6mm hex bolt hard ware.
Available in lengths of 90-140mm in 10mm increments.
cat_picture  <BYTE value>

cat_advert   ProCycle Stem with Pearl Finish
stock_num    106
manu_code    PRC
description  bicycle stem
unit_price   $23.00
unit         each
unit_descr   each
manu_code    PRC
manu_name    ProCycle
lead_time       9
```

The **manu_code** is repeated three times, once for each table, and **stock_num** is repeated twice.

To avoid the considerable duplication of a multiple-table query such as Figure 133: Query on page 1575, include specific columns in the projection list to define the SELECT statement more closely, as the following query shows.

Figure 135. Query

```
SELECT catalog.*, description, unit_price, unit,
      unit_descr, manu_name, lead_time
   FROM catalog, stock, manufact
   WHERE catalog.stock_num = stock.stock_num
       AND stock.manu_code = manufact.manu_code
       AND catalog_num = 10025;
```

The query uses a wildcard to select all columns from the table with the most columns and then specifies columns from the other two tables. The result shows the natural join that the query produces. It displays the same information as the previous example, but without duplication.

Figure 136. Query result

```
catalog_num  10025
stock_num    106
manu_code    PRC
cat_descr
Hard anodized alloy with pearl finish. 6mm hex bolt
hardware. Available in lengths of 90-140mm in 10mm increments.
cat_picture  <BYTE value>

cat_advert   ProCycle Stem with Pearl Finish
description  bicycle stem
unit_price   $23.00
unit         each
unit_descr   each
manu_name    ProCycle
lead_time       9
```

Some query shortcuts

You can use aliases, the INTO TEMP clause, and display labels to speed your way through joins and multiple-table queries and to produce output for other uses.

Aliases

You can assign aliases to the tables in the FROM clause of a SELECT statement to make multiple-table queries shorter and more readable. You can use an alias wherever the table name would be used, for instance, as a prefix to the column names in the other clauses.

Figure 137. Query

```
SELECT s.stock_num, s.manu_code, s.description,
       s.unit_price, c.catalog_num,
       c.cat_advert, m.lead_time
   FROM stock s, catalog c, manufact m
   WHERE s.stock_num = c.stock_num
      AND s.manu_code = c.manu_code
      AND s.manu_code = m.manu_code
      AND s.manu_code IN ('HRO', 'HSK')
      AND s.stock_num BETWEEN 100 AND 301
   ORDER BY catalog_num;
```

The associative nature of the SELECT statement allows you to use an alias before you define it. In the query above, the aliases **s** for the **stock** table, **c** for the **catalog** table, and **m** for the **manufact** table are specified in the FROM clause and used throughout the SELECT and WHERE clauses as column prefixes.

Compare the length of Figure 137: Query on page 1577 with the following query, which does not use aliases.

Figure 138. Query

```
SELECT stock.stock_num, stock.manu_code, stock.description,
       stock.unit_price, catalog.catalog_num,
       catalog.cat_advert,
       manufact.lead_time
   FROM stock, catalog, manufact
   WHERE stock.stock_num = catalog.stock_num
      AND stock.manu_code = catalog.manu_code
      AND stock.manu_code = manufact.manu_code
      AND stock.manu_code IN ('HRO', 'HSK')
      AND stock.stock_num BETWEEN 100 AND 301
   ORDER BY catalog_num;
```

Figure 137: Query on page 1577 and Figure 138: Query on page 1577 are equivalent and retrieve the data that the following query shows.

Figure 139. Query result

```
stock_num    110
manu_code    HRO
description  helmet
unit_price   $260.00
catalog_num  10033
cat_advert   Lightweight Plastic with Vents Assures Cool
             Comfort Without Sacrificing Protection
lead_time       4

stock_num    110
manu_code    HSK
description  helmet
unit_price   $308.00
catalog_num  10034
cat_advert   Teardrop Design Used by Yellow Jerseys; You
             Can Time the Difference
lead_time       5
```

You cannot use the ORDER BY clause for the TEXT column **cat_descr** or the BYTE column **cat_picture**.

You can use aliases to shorten your queries on tables that are not in the current database.

The following query joins columns from two tables that reside in different databases and systems, neither of which is the current database or system.

Figure 140. Query

```
SELECT order_num, lname, fname, phone
FROM masterdb@central:customer c, sales@western:orders o
   WHERE c.customer_num = o.customer_num
      AND order_num <= 1010;
```

By assigning the aliases c and o to the long *database@system:table* names, **masterdb@central:customer** and **sales@western:orders**, respectively, you can use the aliases to shorten the expression in the WHERE clause and retrieve the data, as the result shows.

Figure 141. Query result

```
order_num lname           fname           phone

    1001 Higgins         Anthony         415-368-1100
    1002 Pauli           Ludwig          408-789-8075
    1003 Higgins         Anthony         415-368-1100
    1004 Watson          George          415-389-8789
    1005 Parmelee        Jean            415-534-8822
    1006 Lawson          Margaret        415-887-7235
    1007 Sipes           Arnold          415-245-4578
    1008 Jaeger          Roy             415-743-3611
    1009 Keyes           Frances         408-277-7245
    1010 Grant           Alfred          415-356-1123
```

For more information on how to access tables that are not in the current database, see Access other database servers on page 1696 and the *HCL® Informix® Guide to SQL: Syntax*.

You can also use synonyms as shorthand references to the long names of tables that are not in the current database as well as current tables and views. For details on how to create and use synonyms, see the *IBM® Informix® Database Design and Implementation Guide*.

The INTO TEMP clause

By adding an INTO TEMP clause to your SELECT statement, you can temporarily save the results of a multiple-table query in a separate table that you can query or manipulate without modifying the database. Temporary tables are dropped when you end your SQL session or when your program or report terminates.

The following query creates a temporary table called **stockman** and stores the results of the query in it. Because all columns in a temporary table must have names, the alias **adj_price** is required.

Figure 142. Query

```
SELECT DISTINCT stock_num, manu_name, description,
             unit_price, unit_price * 1.05  adj_price
   FROM stock, manufact
   WHERE manufact.manu_code = stock.manu_code
   INTO TEMP stockman;
SELECT * from stockman;
```

Figure 143. Query result

```
stock_num manu_name       description     unit_price   adj_price

        1 Hero            baseball gloves    $250.00    $262.5000
        1 Husky           baseball gloves    $800.00    $840.0000
        1 Smith           baseball gloves    $450.00    $472.5000
        2 Hero            baseball           $126.00    $132.3000
        3 Husky           baseball bat       $240.00    $252.0000
        4 Hero            football           $480.00    $504.0000
        4 Husky           football           $960.00   $1008.0000

      306 Shimara         tandem adapter     $190.00    $199.5000
      307 ProCycle        infant jogger      $250.00    $262.5000
      308 ProCycle        twin jogger        $280.00    $294.0000
      309 Hero            ear drops           $40.00     $42.0000
      309 Shimara         ear drops           $40.00     $42.0000
      310 Anza            kick board          $84.00     $88.2000
      310 Shimara         kick board          $80.00     $84.0000
      311 Shimara         water gloves        $48.00     $50.4000
      312 Hero            racer goggles       $72.00     $75.6000
      312 Shimara         racer goggles       $96.00    $100.8000
      313 Anza            swim cap            $60.00     $63.0000
      313 Shimara         swim cap            $72.00     $75.6000
```

You can query this table and join it with other tables, which avoids a multiple sort and lets you move more quickly through the database. For more information on temporary tables, see the *HCL® Informix® Guide to SQL: Syntax* and the *HCL® Informix® Administrator's Guide*.

Summary

This chapter presented syntax examples and results for basic kinds of SELECT statements that are used to query a relational database. The section Single-table SELECT statements on page 1538 shows how to perform the following actions:

- Select columns and rows from a table with the Projection and FROM clauses
- Select rows from a table with the Projection, FROM, and WHERE clauses
- Use the DISTINCT or UNIQUE keyword in the Projection clause to eliminate duplicate rows from query results
- Sort retrieved data with the ORDER BY clause and the DESC keyword
- Select and order data values that contain non-English characters
- Use the BETWEEN, IN, MATCHES, and LIKE keywords and various relational operators in the WHERE clause to create comparison conditions
- Create comparison conditions that include values, exclude values, find a range of values (with keywords, relational operators, and subscripting), and find a subset of values
- Use exact-text comparisons, variable-length wildcards, and restricted and unrestricted wildcards to perform variable text searches
- Use the logical operators AND, OR, and NOT to connect search conditions or Boolean expressions in a WHERE clause
- Use the ESCAPE keyword to protect special characters in a query
- Search for NULL values with the IS NULL and IS NOT NULL keywords in the WHERE clause
- Use the FIRST clause to specify that a query returns only a specified number of the rows that match the conditions of the SELECT statement
- Use arithmetic operators in the Projection clause to perform computations on number fields and display derived data
- Assign display labels to computed columns as a formatting tool for reports

This chapter also introduced simple join conditions that enable you to select and display data from two or more tables. The section Multiple-table SELECT statements on page 1568 describes how to perform the following actions:

- Create a Cartesian product
- Create a CROSS JOIN, which creates a Cartesian product
- Include a WHERE clause with a valid join condition in your query to constrain a Cartesian product
- Define and create a natural join and an equi-join
- Join two or more tables on one or more columns
- Use aliases as a shortcut in multiple-table queries
- Retrieve selected data into a separate, temporary table with the INTO TEMP clause to perform computations outside the database

Select data from complex types

This chapter describes how to query *complex data types*. A complex data type is built from a combination of other data types with an SQL type constructor. An SQL statement can access individual components within the complex type. Complex data types are *row types* or *collection types*.

ROW types have instances that combine one or more related data fields. The two kinds of ROW types are *named* and *unnamed*.

Collection types have instances where each collection value contains a group of elements of the same data type, which can be any fundamental or complex data type. A collection can consist of a LIST, SET, or MULTISET datatype.

Important: There is no cross-database support for complex data types. They can only be manipulated in local databases.

For a more complete description of the data types that the database server supports, see the chapter on data types in the *HCL® Informix® Guide to SQL: Reference*.

For information about how to create and use complex types, see the *IBM® Informix® Database Design and Implementation Guide*, *HCL® Informix® Guide to SQL: Reference*, and *HCL® Informix® Guide to SQL: Syntax*.

Select row-type data

This section describes how to query data that is defined as row-type data. A ROW type is a complex type that combines one or more related data fields.

The two kinds of ROW types are as follows:

Named ROW type

A named ROW type can define tables, columns, fields of another row-type column, program variables, statement local variables, and routine return values.

Unnamed ROW type

An unnamed ROW type can define columns, fields of another row-type column, program variables, statement local variables, routine return values, and constants.

The examples used throughout this section use the named ROW types **zip_t**, **address_t**, and **employee_t**, which define the **employee** table. The following figure shows the SQL syntax that creates the ROW types and table.

Figure 144. SQL syntax that creates the ROW types and table.

```
CREATE ROW TYPE zip_t
(
   z_code    CHAR(5),
   z_suffix  CHAR(4)
)

CREATE ROW TYPE address_t
(
   street    VARCHAR(20),
   city      VARCHAR(20),
   state     CHAR(2),
   zip       zip_t
)

CREATE ROW TYPE employee_t
(
name      VARCHAR(30),
address   address_t,
salary    INTEGER
)

CREATE TABLE employee OF TYPE employee_t
```

The named ROW types **zip_t**, **address_t** and **employee_t** serve as templates for the fields and columns of the typed table, **employee**. A *typed table* is a table that is defined on a named ROW type. The **employee_t** type that serves as the template for the **employee** table uses the **address_t** type as the data type of the **address** field. The **address_t** type uses the **zip_t** type as the data type of the **zip** field.

The following figure shows the SQL syntax that creates the **student** table. The **s_address** column of the **student** table is defined on an unnamed ROW type. (The **s_address** column could also have been defined as a named ROW type.)
Figure 145. SQL syntax that creates the student table.

```
CREATE TABLE student
(
s_name      VARCHAR(30),
s_address   ROW(street VARCHAR (20), city VARCHAR(20),
              state CHAR(2), zip VARCHAR(9)),
              grade_point_avg DECIMAL(3,2)
)
```

Select columns of a typed table

A query on a typed table is no different from a query on any other table. For example, the following query uses the asterisk symbol (*) to specify a SELECT statement that returns all columns of the **employee** table.
Figure 146. Query

```
SELECT * FROM employee
```

The SELECT statement on the **employee** table returns all rows for all columns.

Figure 147. Query result

```
name        Paul, J.
address     ROW(102 Ruby, Belmont, CA, 49932, 1000)
salary      78000

name        Davis, J.
address     ROW(133 First, San Jose, CA, 85744, 4900)
salary      75000
⋮
```

The following query shows how to construct a query that returns rows for the **name** and **address** columns of the **employee** table.

Figure 148. Query

```
SELECT name, address FROM employee
```

Figure 149. Query result

```
name        Paul, J.
address     ROW(102 Ruby, Belmont, CA, 49932, 1000)

name        Davis, J.
address     ROW(133 First, San Jose, CA, 85744, 4900)
⋮
```

Select columns that contain row-type data

A *row-type column* is a column that is defined on a named ROW type or unnamed ROW type. You use the same SQL syntax to query a named ROW type and an unnamed row-type column.

A query on a row-type column returns data from all the fields of the ROW type. A *field* is a component data type within a ROW type. For example, the **address** column of the **employee** table contains the **street**, **city**, **state**, and **zip** fields. The following query shows how to construct a query that returns all fields of the **address** column.

Figure 150. Query

```
SELECT address FROM employee
```

Figure 151. Query result

```
address     ROW(102 Ruby, Belmont, CA, 49932, 1000)
address     ROW(133 First, San Jose, CA, 85744, 4900)
address     ROW(152 Topaz, Willits, CA, 69445, 1000))
```

To access individual fields that a column contains, use single-dot notation to project the individual fields of the column. For example, suppose you want to access specific fields from the **address** column of the **employee** table. The following SELECT statement projects the **city** and **state** fields from the **address** column.

Figure 152. Query

```
SELECT address.city, address.state FROM employee
```

Figure 153. Query result

```
city            state

Belmont         CA
San Jose        CA
Willits         CA
```

You construct a query on an unnamed row-type column in the same way you construct a query on a named row-type column. For example, suppose you want to access data from the **s_address** column of the **student** table in Figure 145: SQL syntax that creates the student table. on page 1582. You can use *dot notation* to query the individual fields of a column that are defined on an unnamed row type. The following query shows how to construct a SELECT statement on the **student** table that returns rows for the **city** and **state** fields of the **s_address** column.

Figure 154. Query

```
SELECT s_address.city, s_address.state FROM student
```

Figure 155. Query result

```
city            state

Belmont         CA
Mount Prospect  IL
Greeley         CO
```

Field projections

Do not confuse fields with columns. Columns are only associated with tables, and column projections use conventional dot notation of the form `name_1.name2` for a table and column, respectively. A *field* is a component data type within a ROW type. With ROW types (and the capability to assign a ROW type to a single column), you can project individual fields of a column with single dot notation of the form: `name_a.name_b.name_c.name_d`. HCL® Informix® database servers use the following precedence rules to interpret dot notation:

1. table_name_a . column_name_b . field_name_c . field_name_d
2. column_name_a . field_name_b . field_name_c . field_name_d

When the meaning of a particular identifier is ambiguous, the database server uses precedence rules to determine which database object the identifier specifies. Consider the following two statements:

```
CREATE TABLE b (c ROW(d INTEGER, e CHAR(2)))
CREATE TABLE c (d INTEGER)
```

In the following SELECT statement, the expression `c.d` references column **d** of table **c** (rather than field **d** of column **c** in table **b**) because a table identifier has a higher precedence than a column identifier:

```
SELECT * FROM b,c WHERE c.d = 10
```

To avoid referencing the wrong database object, you can specify the full notation for a field projection. Suppose, for example, you want to reference field **d** of column **c** in table **b** (not column **d** of table **c**). The following statement specifies the table, column, and field identifiers of the object you want to reference:

```
SELECT * FROM b,c WHERE b.c.d = 10
```

Important: Although precedence rules reduce the chance of the database server misinterpreting field projections, it is recommended that you use unique names for all table, column, and field identifiers.

Field projections to select nested fields

Typically the row type is a column, but you can use any row-type expression for field projection. When the row-type expression itself contains other row types, the expression contains nested fields. To access nested fields within an expression or individual fields, use dot notation. To access all the fields of the row type, use an asterisk (*). This section describes both methods of row-type access.

For a discussion of how to use dot notation and asterisk notation with row-type expressions, see the Expression segment in the *HCL® Informix® Guide to SQL: Syntax*.

Select individual fields of a row type

Consider the **address** column of the **employee** table, which contains the fields **street**, **city**, **state**, and **zip**. In addition, the **zip** field contains the nested fields: **z_code** and **z_suffix**. (You might want to review the row type and table definitions of Figure 144: SQL syntax that creates the ROW types and table. on page 1582.) A query on the **zip** field returns rows for the **z_code** and **z_suffix** fields. However, you can specify that a query returns only specific nested fields. The following query shows how to use dot notation to construct a SELECT statement that returns rows for the **z_code** field of the **address** column only.

Figure 156. Query

```
SELECT address.zip.z_code FROM employee
```

Figure 157. Query result

```
z_code

39444
6500
76055
19004
⋮
```

Asterisk notation to access all fields of a row type

Asterisk notation is supported only within the select list of a SELECT statement. When you specify the column name for a row-type column in a projection list, the database server returns values for all fields of the column. You can also use asterisk notation when you want to project all the fields within a ROW type.

The following query uses asterisk notation to return all fields of the **address** column in the **employee** table.

Figure 158. Query

```
SELECT address.* FROM employee;
```

Figure 159. Query result

```
address   ROW(102 Ruby, Belmont, CA, 49932, 1000)
address   ROW(133 First, San Jose, CA, 85744, 4900)
address   ROW(152 Topaz, Willits, CA, 69445, 1000))
⋮
```

The asterisk notation makes it easier to perform some SQL tasks. Suppose you create a function new_row() that returns a row-type value and you want to call this function and insert the row that is returned into a table. The database server provides no easy way to handle such operations. However, the following query shows how to use asterisk notation to return all fields of new_row() and insert the returned fields into the **tab_2** table.

Figure 160. Query

```
INSERT INTO tab_2 SELECT new_row(exp).* FROM tab_1
```

For information about how to use the INSERT statement, see Modify data on page 1658.

Important: An expression that uses the `.*` notation is evaluated only once.

Select from a collection

This section describes how to query columns that are defined on collection types. A *collection type* is a complex data type in which each collection value contains a group of elements of the same data type. For a detailed description of collection data types, see the *IBM® Informix® Database Design and Implementation Guide*. For information about how to access the individual elements that a collection contains, see Handle collections in SELECT statements on page 1645.

The following figure shows the **manager** table, which is used in examples throughout this section. The **manager** table contains both simple and nested collection types. A *simple collection* is a collection type that does not contain any fields that are themselves collection types. The **direct_reports** column of the **manager** table is a simple collection. A *nested collection* is a collection type that contains another collection type. The **projects** column of the **manager** table is a nested collection.

Figure 161. The manager table

```
CREATE TABLE manager
(
   mgr_name        VARCHAR(30),
   department      VARCHAR(12),
   direct_reports  SET(VARCHAR(30) NOT NULL),
   projects        LIST(ROW(pro_name VARCHAR(15),
                     pro_members SET(VARCHAR(20) NOT NULL)
                     ) NOT NULL)
)
```

A query on a column that is a collection type returns, for each row in the table, all the elements that the particular collection contains. For example, the following query shows a query that returns data in the **department** column and all elements in the **direct_reports** column for each row of the **manager** table.

Figure 162. Query

```
SELECT department, direct_reports FROM manager
```

Figure 163. Query result

```
department      marketing
direct_reports  SET {Smith, Waters, Adams, Davis, Kurasawa}

department      engineering
ddirect_reports  SET {Joshi, Davis, Smith, Waters, Fosmire, Evans, Jones}

department      publications
direct_reports  SET {Walker, Fremont, Porat, Johnson}

department      accounting
direct_reports  SET {Baker, Freeman, Jacobs}
```

The output of a query on a collection type always includes the type constructor that specifies whether the collection is a SET, MULTISET, or LIST. For example, in the result, the SET constructor precedes the elements of each collection. Braces ({}) demarcate the elements of a collection; commas separate individual elements of a collection.

Select nested collections

The **projects** column of the **manager** table (see Figure 161: The manager table on page 1586) is a nested collection. A query on a nested collection type returns all the elements that the particular collection contains. The following query shows a query that returns all elements from the **projects** column for a specified row. The WHERE clause limits the query to a single row in which the value in the **mgr_name** column is `Sayles`.

Figure 164. Query

```
SELECT projects
   FROM manager
   WHERE mgr_name = 'Sayles'
```

The query result shows a **project** column collection for a single row of the **manager** table. The query returns the names of those projects that the manager `Sayles` oversees. The collection contains, for each element in the LIST, the project name (**pro_name**) and the SET of individuals (**pro_members**) who are assigned to each project.

Figure 165. Query result

```
projects  LIST {ROW(voyager_project, SET{Simonian, Waters, Adams, Davis})}

projects  LIST {ROW(horizon_project, SET{Freeman, Jacobs, Walker, Cannan})}

projects  LIST {ROW(sapphire_project, SET{Villers, Reeves, Doyle, Strongin})}
⋮
```

The IN keyword to search for elements in a collection

You can use the IN keyword in the WHERE clause of an SQL statement to determine whether a collection contains a certain element. For example, the following query shows how to construct a query that returns values for **mgr_name** and **department** where `Adams` is an element of a collection in the **direct_reports** column.

Figure 166. Query

```
SELECT mgr_name, department
   FROM manager
   WHERE 'Adams' IN direct_reports
```

Figure 167. Query result

```
mgr_name     Sayles
department   marketing
```

Although you can use a WHERE clause with the IN keyword to search for a particular element in a simple collection, the query always returns the complete collection. For example, the following query returns all the elements of the collection where `Adams` is an element of a collection in the **direct_reports** column.

Figure 168. Query

```
SELECT mgr_name, direct_reports
   FROM manager
   WHERE 'Adams' IN direct_reports
```

Figure 169. Query result

```
mgr_name         Sayles
direct_reports  SET {Smith, Waters, Adams, Davis, Kurasawa}
```

As the result shows, a query on a collection column returns the entire collection, not a particular element within the collection.

You can use the IN keyword in a WHERE clause to reference a simple collection only. You cannot use the IN keyword to reference a collection that contains fields that are themselves collections. For example, you cannot use the IN keyword to reference the **projects** column in the **manager** table because **projects** is a nested collection.

You can combine the NOT and IN keywords in the WHERE clause of a SELECT statement to search for collections that do not contain a certain element. For example, the following query shows a query that returns values for **mgr_name** and **department** where `Adams` is not an element of a collection in the **direct_reports** column.

Figure 170. Query

```
SELECT mgr_name, department
   FROM manager
   WHERE 'Adams' NOT IN direct_reports
```

Figure 171. Query result

```
mgr_name     Williams
department   engineering

mgr_name     Lyman
department   publications

mgr_name     Cole
department   accounting
```

For information about how to count the elements in a collection column, see Cardinality function on page 1603.

Select rows within a table hierarchy

This section describes how to query rows from tables within a table hierarchy. For more information about how to create and use a table hierarchy, see the *IBM® Informix® Database Design and Implementation Guide*.

The following figure shows the statements that create the type and table hierarchies that the examples in this section use.
Figure 172. Statements that create the type and table hierarchies.

```
CREATE ROW TYPE address_t
(
   street   VARCHAR (20),
   city     VARCHAR(20),
   state    CHAR(2),
   zip      VARCHAR(9)
)

CREATE ROW TYPE person_t
(
   name     VARCHAR(30),
   address  address_t,
   soc_sec  CHAR(9)
)

CREATE ROW TYPE employee_t
(
salary       INTEGER
)
UNDER person_t

CREATE ROW TYPE sales_rep_t
(
   rep_num     SERIAL8,
   region_num  INTEGER
)
UNDER employee_t

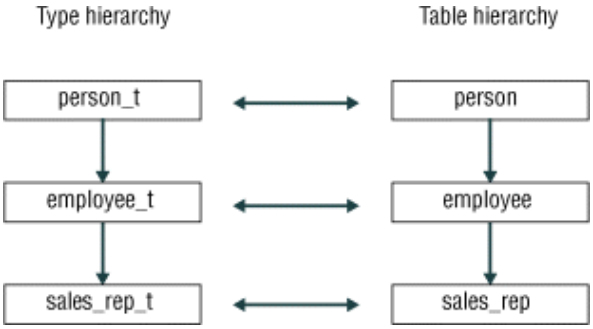
CREATE TABLE person OF TYPE person_t

CREATE TABLE employee OF TYPE employee_t
UNDER person

CREATE TABLE sales_rep OF TYPE sales_rep_t
UNDER employee
```

The following figure shows the hierarchical relationships of the row types and tables in the previous figure.

Figure 173. Type and table hierarchies



Select rows of the supertable without the ONLY keyword

A table hierarchy allows you to construct, in a single SQL statement, a query whose scope is a supertable and its subtables. A query on a supertable returns rows from both the supertable and its subtables. The following query shows a query on the **person** table, which is the root supertable in the table hierarchy.

Figure 174. Query

```
SELECT * FROM person
```

Figure 55: Query result on page 1544 returns all columns in the supertable and those columns in subtables (**employee** and **sales_rep**) that are inherited from the supertable. A query on a supertable does not return columns from subtables that are not in the supertable. The query result shows the **name**, **address**, and **soc_sec** columns in the **person**, **employee**, and **sales_rep** tables.

Figure 175. Query result

```
name        Rogers, J.
address     ROW(102 Ruby Ave, Belmont, CA, 69055)
soc_sec     454849344

name        Sallie, A.
address     ROW(134 Rose St, San Carlos, CA, 69025)
soc_sec     348441214
⋮
```

Select rows from a supertable with the ONLY keyword

Although a SELECT statement on a supertable returns rows from both the supertable and its subtables, you cannot tell which rows come from the supertable and which rows come from the subtables. To limit the results of a query to the supertable only, you must include the ONLY keyword in the SELECT statement. For example, the following query returns rows in the **person** table only.

Figure 176. Query

```
SELECT * FROM ONLY(person);
```

Figure 177. Query result

```
name        Rogers, J.
address     ROW(102 Ruby Ave, Belmont, CA, 69055)
soc_sec     454849344
⋮
```

An alias for a supertable

An *alias* is a word that immediately follows the name of a table in the FROM clause. You can specify an alias for a typed table in a SELECT or UPDATE statement and then use the alias (in the same SELECT or UPDATE statement) as an expression by itself. If you create an alias for a supertable, the alias can represent values from the supertable or the subtables that inherit from the supertable. In , the following query returns row values for all instances of the **person**, **employee**, and **sales_rep** tables.

Figure 178. Query

```
SELECT p FROM person p;
```

does not recognize this construct. In programs, the query returns an error.

Summary

This chapter introduced sample syntax and results for selecting data from complex types using SELECT statements to query a relational database. The section Select row-type data on page 1581 shows how to perform the following actions:

- Select row-type data from typed tables and columns
- Use row-type expressions for field projections

The section Select from a collection on page 1586 shows how to perform the following actions:

- Query columns that are defined on collection types
- Search for elements in a collection
- Query columns that are defined on nested collection types

The section Select rows within a table hierarchy on page 1589 shows how to perform the following actions:

- Query a supertable with or without the ONLY keyword
- Specify an alias for a supertable

Functions in SELECT statements

In addition to column names and operators, an expression can also include one or more functions. This chapter shows how to use functions in SELECT statements to perform more complex database queries and data manipulation.

For information about the syntax of the following SQL functions and other SQL functions, see the Expressions segment in the *HCL® Informix® Guide to SQL: Syntax*.

Tip: You can also use functions that you create yourself. For information about user-defined functions, see Create and use SPL routines on page 1753, and *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

Functions in SELECT statements

You can use any basic type of expression (column, constant, function, aggregate function, and procedure), or combination thereof, in the select list.

A function expression uses a function that is evaluated for each row in the query. All function expressions require arguments. This set of expressions contains the time function and the length function when they are used with a column name as an argument.

Aggregate functions

An aggregate function returns one value for a set of queried rows. The aggregate functions take on values that depend on the set of rows that the WHERE clause of the SELECT statement returns. In the absence of a WHERE clause, the aggregate functions take on values that depend on all the rows that the FROM clause forms.

You cannot use aggregate functions for expressions that contain the following data types:

- TEXT
- BYTE
- CLOB
- BLOB
- Collection data types (LIST, MULTISET, and SET)
- ROW types
- Opaque data types (except with user-defined aggregate functions that support opaque types)

Aggregates are often used to summarize information about groups of rows in a table. This use is discussed in Compose advanced SELECT statements on page 1619. When you apply an aggregate function to an entire table, the result contains a single row that summarizes all the selected rows.

All HCL® Informix® database servers support the following aggregate functions.

The AVG function

The following query computes the average **unit_price** of all rows in the **stock** table.

Figure 179. Query

```
SELECT AVG (unit_price) FROM stock;
```

Figure 180. Query result

```
  (avg)

$197.14
```

The following query computes the average **unit_price** of just those rows in the **stock** table that have a **manu_code** of `SHM`.

Figure 181. Query

```
SELECT AVG (unit_price) FROM stock WHERE manu_code = 'SHM';
```

Figure 182. Query result

```
  (avg)

$204.93
```

The COUNT function

The following query counts and displays the total number of rows in the **stock** table.

Figure 183. Query

```
SELECT COUNT(*) FROM stock;
```

Figure 184. Query result

```
(count(*))

   73
```

The following query includes a WHERE clause to count specific rows in the **stock** table, in this case, only those rows that have a **manu_code** of `SHM`.

Figure 185. Query

```
SELECT COUNT (*) FROM stock WHERE manu_code = 'SHM';
```

Figure 186. Query result

```
(count(*))

   17
```

By including the keyword DISTINCT (or its synonym UNIQUE) and a column name in the following query, you can tally the number of different manufacturer codes in the **stock** table.

Figure 187. Query

```
SELECT COUNT (DISTINCT manu_code) FROM stock;
```

Figure 188. Query result

```
(count)

   9
```

The MAX and MIN functions

You can combine aggregate functions in the same SELECT statement. For example, you can include both the MAX and the MIN functions in the select list, as the following query shows.

Figure 189. Query

```
SELECT MAX (ship_charge), MIN (ship_charge) FROM orders;
```

The query finds and displays both the highest and lowest **ship_charge** in the **orders** table.

Figure 190. Query result

```
 (max)     (min)

$25.20     $5.00
```

The RANGE function

The RANGE function computes the difference between the maximum and the minimum values for the selected rows.

You can apply the RANGE function only to numeric columns. The following query finds the range of prices for items in the **stock** table.

Figure 191. Query

```
SELECT RANGE(unit_price) FROM stock;
```

Figure 192. Query result

```
(range)

955.50
```

As with other aggregates, the RANGE function applies to the rows of a group when the query includes a GROUP BY clause, which the following query shows.

Figure 193. Query

```
SELECT RANGE(unit_price) FROM stock
   GROUP BY manu_code;
```

Figure 194. Query result

```
(range)

820.20
595.50
720.00
225.00
632.50
  0.00
460.00
645.90
425.00
```

The STDEV function

The STDEV function computes the standard deviation for the column of the selected rows without *Bessel correction*. It has the aliases STDEV_POP, STDDEV and STDDEV_POP - the "pop" part of the name indicating that this is the population standard deviation. It is returning the square root of the VARIANCE of the selected columns.

You can apply the STDEV function only to numeric columns. The following query finds the standard deviation on a population:

```
SELECT STDEV(age) FROM u_pop WHERE age > 21;
```

As with the other aggregates, the STDEV function applies to the rows of a group when the query includes a GROUP BY clause, as the following example shows:

```
SELECT STDEV(age) FROM u_pop
   GROUP BY state
   WHERE STDEV(age) > 21;
```

Nulls are ignored unless every value in the specified column is null. If every column value is null, the STDEV function returns a null for that column. For more information about the STDEV function, see the Expression segment in the *HCL® Informix® Guide to SQL: Syntax*.

The SUM function

The following query calculates the total **ship_weight** of orders that were shipped on July 13, 1998.

Figure 195. Query

```
SELECT SUM (ship_weight) FROM orders
   WHERE ship_date = '07/13/1998';
```

Figure 196. Query result

```
(sum)

130.5
```

The VARIANCE function

The VARIANCE function returns the variance for the column of the selected rows without Bessel correction. This is the population variance. The function has an according alias VARIANCE_POP. It computes the following value:

```
(SUM(Xi**2) - (SUM(Xi)**2)/N)/(N-1)
```

In this example, `Xi` is each value in the column and `N` is the total number of values in the column. You can apply the VARIANCE function only to numeric columns. The following query finds the variance on a population:

```
SELECT VARIANCE(age) FROM u_pop WHERE age > 21;
```

As with the other aggregates, the VARIANCE function applies to the rows of a group when the query includes a GROUP BY clause, which the following example shows:

```
SELECT VARIANCE(age) FROM u_pop
   GROUP BY birth
   WHERE VARIANCE(age) > 21;
```

Nulls are ignored unless every value in the specified column is null. If every column value is null, the VARIANCE function returns a null for that column. For more information about the VARIANCE function, see the Expression segment in the *HCL® Informix® Guide to SQL: Syntax*.

Apply aggregate functions to expressions

The following query shows how you can apply aggregate functions to arithmetic expressions, and declare display labels for their results:

Figure 197. Query

```
SELECT MAX (res_dtime - call_dtime) maximum,
   MIN (res_dtime - call_dtime) minimum,
   AVG (res_dtime - call_dtime) average
   FROM cust_calls;
```

The query finds and displays the maximum, minimum, and average amounts of time (in days, hours, and minutes) between the reception and resolution of a customer call, and labels the derived values appropriately. The query result shows these aggregate time-interval values that the query calculates:

Figure 198. Query result

```
maximum         minimum         average

5 20:55         0 00:01         1 02:56
```

Time functions

You can use the time functions DAY, MONTH, WEEKDAY, and YEAR in either the Projection clause or the WHERE clause of a query. These functions return a value that corresponds to the expressions or arguments that you use to call the function. You can also use the CURRENT or SYSDATE function to return a value with the current date and time, or use the **EXTEND** function to adjust the precision of a DATE or DATETIME value.

The DAY and CURRENT functions

The following query returns the day of the month for the **call_dtime** and **res_dtime** columns in two *expression* columns.

Figure 199. Query

```
SELECT customer_num, DAY (call_dtime), DAY (res_dtime)
   FROM cust_calls;
```

Figure 200. Query result

```
customer_num  (expression) (expression)

         106            12          12
         110             7           7
         119             1           2
         121            10          10
         127            31
         116            28          28
         116            21          27
```

The following query uses the DAY and CURRENT functions to compare column values to the current day of the month. It selects only those rows where the value is earlier than the current day. In this example, the CURRENT day is `15`.

Figure 201. Query

```
SELECT customer_num, DAY (call_dtime), DAY (res_dtime)
   FROM cust_calls
   WHERE DAY (call_dtime) < DAY (CURRENT);
```

Figure 202. Query result

```
customer_num  (expression) (expression)
         106           12           12
         110            7            7
         119            1            2
         121           10           10
```

The following query uses the CURRENT function to select all calls except those that came in today.

Figure 203. Query

```
SELECT customer_num, call_code, call_descr
   FROM cust_calls
   WHERE call_dtime < CURRENT YEAR TO DAY;
```

Figure 204. Query result

```
customer_num  106
call_code     D
call_descr    Order was received, but two of the cans of ANZ tennis balls
              within the case were empty

customer_num  110
call_code     L
call_descr     Order placed one month ago (6/7) not received.

customer_num  116
call_code     I
call_descr    Second complaint from this customer! Received two cases
              right-handed outfielder gloves (1 HRO) instead of one case
              lefties.
```

The SYSDATE function closely resembles the CURRENT function, but the default precision of its returned value is DATETIME YEAR TO FRACTION(5), rather than the default DATETIME YEAR TO FRACTION(3) precision of CURRENT when no DATETIME qualifier is specified.

The MONTH function

The following query uses the MONTH function to extract and show what month the customer call was received and resolved, and it uses display labels for the resulting columns. However, it does not make a distinction between years.

Figure 205. Query

```
SELECT customer_num,
   MONTH (call_dtime) call_month,
   MONTH (res_dtime) res_month
   FROM cust_calls;
```

Figure 206. Query result

```
customer_num   call_month    res_month

         106            6            6
         110            7            7
         119            7            7
         121            7            7
         127            7
         116           11           11
         116           12           12
```

The following query uses the MONTH function plus DAY and CURRENT to show what month the customer call was received and resolved if DAY is earlier than the current day.

Figure 207. Query

```
SELECT customer_num,
   MONTH (call_dtime) called,
   MONTH (res_dtime) resolved
   FROM cust_calls
   WHERE DAY (res_dtime) < DAY (CURRENT);
```

Figure 208. Query result

```
customer_num     called   resolved

         106          6          6
         119          7          7
         121          7          7
```

The WEEKDAY function

The following query uses the WEEKDAY function to indicate which day of the week calls are received and resolved (`0` represents Sunday, `1` is Monday, and so on), and the expression columns are labeled.

Figure 209. Query

```
SELECT customer_num,
   WEEKDAY (call_dtime) called,
   WEEKDAY (res_dtime) resolved
   FROM cust_calls
   ORDER BY resolved;
```

Figure 210. Query result

```
customer_num     called   resolved

         127          3
         110          0          0
         119          1          2
         121          3          3
         116          3          3
         106          3          3
         116          5          4
```

The following query uses the COUNT and WEEKDAY functions to count how many calls were received on a weekend. This kind of statement can give you an idea of customer-call patterns or indicate whether overtime pay might be required.

Figure 211. Query

```
SELECT COUNT(*)
   FROM cust_calls
   WHERE WEEKDAY (call_dtime) IN (0,6);
```

Figure 212. Query result

```
(count(*))

         4
```

The YEAR function

The following query retrieves rows where the **call_dtime** is earlier than the beginning of the current year.

Figure 213. Query

```
SELECT customer_num, call_code,
   YEAR (call_dtime) call_year,
   YEAR (res_dtime) res_year
   FROM cust_calls
   WHERE YEAR (call_dtime) < YEAR (TODAY);
```

Figure 214. Query result

```
customer_num call_code call_year res_year

         116 I              1997     1997
         116 I              1997     1997
```

Format DATETIME values

In the following query, the EXTEND function displays only the specified subfields to restrict the two DATETIME values.

Figure 215. Query

```
SELECT customer_num,
   EXTEND (call_dtime, month to minute) call_time,
   EXTEND (res_dtime, month to minute) res_time
   FROM cust_calls
   ORDER BY res_time;
```

The query returns the month-to-minute range for the columns labeled **call_time** and **res_time** and gives an indication of the work load.

Figure 216. Query result

```
customer_num call_time   res_time

         127 07-31 14:30
         106 06-12 08:20 06-12 08:25
         119 07-01 15:00 07-02 08:21
         110 07-07 10:24 07-07 10:30
         121 07-10 14:05 07-10 14:06
         116 11-28 13:34 11-28 16:47
         116 12-21 11:24 12-27 08:19
```

The TO_CHAR function can also format DATETIME values. See The TO_CHAR function on page 1601 for information about this built-in function, which can also accept DATE values or numeric values as an argument, and returns a formatted character string.

Besides the built-in time functions that these examples illustrate, HCL Informix® also supports the ADD_MONTHS, LAST_DAY, MDY, MONTHS_BETWEEN, NEXT_DAY, and QUARTER functions. In addition to these functions, the TRUNC and ROUND functions can return values that change the precision of DATE or DATETIME arguments. These additional time functions are described in the *HCL® Informix® Guide to SQL: Syntax*.

Date-conversion functions

You can use a date-conversion function anywhere you use an expression.

The following conversion functions convert between date and character values:

The DATE function

The DATE function converts a character string to a DATE value. In the following query, the DATE function converts a character string to a DATE value to allow for comparisons with DATETIME values. The query retrieves DATETIME values only when **call_dtime** is later than the specified DATE.

Figure 217. Query

```
SELECT customer_num, call_dtime, res_dtime
   FROM cust_calls
   WHERE call_dtime > DATE ('12/31/97');
```

Figure 218. Query result

```
customer_num call_dtime       res_dtime

         106 1998-06-12 08:20 1998-06-12 08:25
         110 1998-07-07 10:24 1998-07-07 10:30
         119 1998-07-01 15:00 1998-07-02 08:21
         121 1998-07-10 14:05 1998-07-10 14:06
         127 1998-07-31 14:30
```

The following query converts DATETIME values to DATE format and displays the values, with labels, only when **call_dtime** is greater than or equal to the specified date.

Figure 219. Query

```
SELECT customer_num,
   DATE (call_dtime) called,
   DATE (res_dtime) resolved
   FROM cust_calls
   WHERE call_dtime >= DATE ('1/1/98');
```

Figure 220. Query result

```
customer_num called       resolved

         106 06/12/1998   06/12/1998
         110 07/07/1998   07/07/1998
         119 07/01/1998   07/02/1998
         121 07/10/1998   07/10/1998
         127 07/31/1998
```

The TO_CHAR function

The TO_CHAR function converts DATETIME or DATE values to character string values. The TO_CHAR function evaluates a DATETIME value according to the date-formatting directive that you specify and returns an NVARCHAR value. For a complete list of the supported date-formatting directives, see the description of the **GL_DATETIME** environment variable in the *HCL® Informix® GLS User's Guide*.

You can also use the TO_CHAR function to convert a DATETIME or DATE value to an LVARCHAR value.

The following query uses the TO_CHAR function to convert a DATETIME value to a more readable character string.

Figure 221. Query

```
SELECT customer_num,
   TO_CHAR(call_dtime, "%A %B %d %Y") call_date
   FROM cust_calls
   WHERE call_code = "B";
```

Figure 222. Query result

```
customer_num  119
call_date     Friday July 01 1998
```

The following query uses the TO_CHAR function to convert DATE values to more readable character strings.

Figure 223. Query

```
SELECT order_num,
   TO_CHAR(ship_date,"%A %B %d %Y") date_shipped
   FROM orders
   WHERE paid_date IS NULL;
```

Figure 224. Query result

```
order_num            1004
date_shipped         Monday May 30 1998

order_num            1006
date_shipped

order_num            1007
date_shipped         Sunday June 05 1998

order_num            1012
date_shipped         Wednesday June 29 1998

order_num            1016
date_shipped         Tuesday July 12 1998

order_num            1017
date_shipped         Wednesday July 13 1998
```

The TO_CHAR function can also format numeric values. For more information about the built-in TO_CHAR function, see the *HCL® Informix® Guide to SQL: Syntax*.

The TO_DATE function

The TO_DATE function accepts an argument of a character data type and converts this value to a DATETIME value. The TO_DATE function evaluates a character string according to the date-formatting directive that you specify and returns a DATETIME value. For a complete list of the supported date-formatting directives, see the description of the **GL_DATETIME** environment variable in the *HCL® Informix® GLS User's Guide*.

You can also use the TO_DATE function to convert an LVARCHAR value to a DATETIME value.

The following query uses the TO_DATE function to convert character string values to DATETIME values whose format you specify.

Figure 225. Query

```
SELECT customer_num, call_descr
  FROM cust_calls
  WHERE call_dtime = TO_DATE("2008-07-07 10:24",
  "%Y-%m-%d %H:%M");
```

Figure 226. Query result

```
customer_num   110

call_descr     Order placed one month ago (6/7) not received.
```

You can use the DATE or TO_DATE function to convert a character string to a DATE value. One advantage of the TO_DATE function is that it allows you to specify a format for the value returned. (You can use the TO_DATE function, which always returns a DATETIME value, to convert a character string to a DATE value because the database server implicitly handles conversions between DATE and DATETIME values.)

Cardinality function

The CARDINALITY function counts the number of elements that a collection contains. You can use the CARDINALITY function with simple or nested collections. Any duplicates in a collection are counted as individual elements. The following query shows a query that returns, for every row in the **manager** table, **department** values and the number of elements in each **direct_reports** collection.

Figure 227. Query

```
SELECT department, CARDINALITY(direct_reports) FROM manager;
```

Figure 228. Query result

```
department   marketing 5

department   engineering 7

department   publications 4

department   accounting 3
```

You can also evaluate the number of elements in a collection from within a predicate expression, as the following query shows.

Figure 229. Query

```
SELECT department, CARDINALITY(direct_reports) FROM manager
   WHERE CARDINALITY(direct_reports) < 6
   GROUP BY department;
```

Figure 230. Query result

```
department   accounting 3

department   marketing 5

department   publications 4
```

Smart large object functions

The database server provides four SQL functions that you can call from within an SQL statement to import and export smart large objects. The following table shows the smart-large-object functions.

Table 84. SQL functions for smart large objects

Function name	Purpose
FILETOBLOB()	Copies a file into a BLOB column
FILETOCLOB()	Copies a file into a CLOB column
LOCOPY()	Copies BLOB or CLOB data into another BLOB or CLOB column
LOTOFILE()	Copies a BLOB or CLOB into a file

For detailed information and the syntax of smart-large-object functions, see the Expression segment in the *HCL® Informix® Guide to SQL: Syntax*.

You can use any of the functions that the table shows in SELECT, UPDATE, and INSERT statements. For examples of how to use the preceding functions in INSERT and UPDATE statements, see Modify data on page 1658.

Suppose you create the **inmate** and **fbi_list** tables, as the following figure shows.

Figure 231. Create the inmate and fbi_list tables.

```
CREATE TABLE inmate
(
   id_num   INT,
   picture  BLOB,
   felony   CLOB
);

CREATE TABLE fbi_list
(
   id       INTEGER,
   mugshot  BLOB
) PUT mugshot IN (sbspace1);
```

The following SELECT statement uses the LOTOFILE() function to copy data from the **felony** column into the `felon_322.txt` file that is located on the client computer:

```
SELECT id_num, LOTOFILE(felony, 'felon_322.txt', 'client')
   FROM inmate
   WHERE id = 322;
```

The first argument for LOTOFILE() specifies the name of the column from which data is to be exported. The second argument specifies the name of the file into which data is to be copied. The third argument specifies whether the target file is located on the client computer ('client') or server computer ('server').

The following rules apply for specifying the path of a file name in a function argument, depending on whether the file resides on the client or server computer:

- If the source file resides on the server computer, you must specify the full path name to the file (not the path name relative to the current working directory).
- If the source file resides on the client computer, you can specify either the full or relative path name to the file.

String-manipulation functions

String-manipulation functions accept arguments of type CHAR, NCHAR, VARCHAR, NVARCHAR, or LVARCHAR. You can use a string-manipulation function anywhere you use an expression.

The following functions convert between upper and lowercase letters in a character string:

- LOWER
- UPPER
- INITCAP

The following functions manipulate character strings in various ways:

- REPLACE
- SUBSTR
- SUBSTRING
- LPAD
- RPAD

Restriction: You cannot overload any of the string-manipulation functions to handle extended data types.

The LOWER function

Use the LOWER function to replace every uppercase letter in a character string with a lowercase letter. The LOWER function accepts an argument of a character data type and returns a value of the same data type as the argument you specify.

The following query uses the LOWER function to convert any uppercase letters in a character string to lowercase letters.

Figure 232. Query

```
SELECT manu_code, LOWER(manu_code)
   FROM items
   WHERE order_num = 1018
```

Figure 233. Query result

```
manu_code   (expression)

PRC          prc
KAR          kar
PRC          prc
SMT          smt
HRO          hro
```

The UPPER function

Use the UPPER function to replace every lowercase letter in a character string with an uppercase letter. The UPPER function accepts an argument of a character data type and returns a value of the same data type as the argument you specify.

The following query uses the UPPER function to convert any lowercase letters in a character string to uppercase letters.

Figure 234. Query

```
SELECT call_code, UPPER(code_descr) FROM call_type
```

Figure 235. Query result

```
call_code    (expression)

B            BILLING ERROR
D            DAMAGED GOODS
I            INCORRECT MERCHANDISE SENT
L            LATE SHIPMENT
O            OTHER
```

The INITCAP function

Use the INITCAP function to replace the first letter of every word in a character string with an uppercase letter. The INITCAP function assumes a new word whenever the function encounters a letter that is preceded by any character other than a letter. The INITCAP function accepts an argument of a character data type and returns a value of the same data type as the argument you specify.

The following query uses the INITCAP function to convert the first letter of every word in a character string to an uppercase letter.

Figure 236. Query

```
SELECT INITCAP(description) FROM stock
   WHERE manu_code = "ANZ";
```

Figure 237. Query result

```
(expression)

Tennis Racquet
Tennis Ball
Volleyball
Volleyball Net
Helmet
Golf Shoes
3 Golf Balls
Running Shoes
Watch
Kick Board
Swim Cap
```

The REPLACE function

Use the REPLACE function to replace a certain set of characters in a character string with other characters.

In the following query, the REPLACE function replaces the unit column value `each` with `item` for every row that the query returns. The first argument of the REPLACE function is the expression to be evaluated. The second argument specifies the characters that you want to replace. The third argument specifies a new character string to replace the characters removed.

Figure 238. Query

```
SELECT stock_num, REPLACE(unit,"each", "item") cost_per, unit_price
   FROM stock
   WHERE manu_code = "HRO";
```

Figure 239. Query result

```
stock_num      cost_per   unit_price

1              case          $250.00
2              case          $126.00
4              case          $480.00
7              case          $600.00
110            case          $260.00
205            case          $312.00
301            item           $42.50
302            item            $4.50
304            box           $280.00
305            case           $48.00
309            case           $40.00
312            box            $72.00
```

The SUBSTRING and SUBSTR functions

You can use the SUBSTRING and SUBSTR functions to return a portion of a character string. You specify the *start position* and *length* (optional) to determine which portion of the character string the function returns.

Restriction: The units of measurement in the arguments to these two functions are bytes, rather than logical characters. This is of no importance in the default locale, nor in other single-byte locales, but you should not invoke SUBSTRING or SUBSTR in locales in which the logical characters of the code set can differ in their storage lengths.

The SUBSTRING function

You can use the SUBSTRING function to return some portion of a character string. You specify the *start position* and *length* (optional) to determine which portion of the character string the function returns. You can specify a positive or negative number for the start position. A start position of `1` specifies that the SUBSTRING function begins from the first position in the string. When the start position is zero (0) or a negative number, the SUBSTRING function counts backward from the beginning of the string.

The following query shows an example of the SUBSTRING function, which returns the first four characters for any **sname** column values that the query returns. In this example, the SUBSTRING function starts at the beginning of the string and returns four characters counting forward from the start position.

Figure 240. Query

```
SELECT sname, SUBSTRING(sname FROM 1 FOR 4) FROM state
   WHERE code = "AZ";
```

Figure 241. Query result

```
sname          (expression)

Arizona        Ariz
```

In the following query, the SUBSTRING function specifies a start position of `6` but does not specify the length. The function returns a character string that extends from the sixth position to the end of the string.

Figure 242. Query

```
SELECT sname, SUBSTRING(sname FROM 6) FROM state
   WHERE code = "WV";
```

Figure 243. Query result

```
sname          (expression)

West Virginia   Virginia
```

In the following query, the SUBSTRING function returns only the first character for any **sname** column value that the query returns. For the SUBSTRING function, a start position of `-2` counts backward three positions (`0, -1, -2`) from the start position of the string (for a start position of `0`, the function counts backward one position from the beginning of the string).

Figure 244. Query

```
SELECT sname, SUBSTRING(sname FROM -2 FOR 4) FROM state
   WHERE code = "AZ";
```

Figure 245. Query result

```
sname          (expression)

Arizona        A
```

The SUBSTR function

The SUBSTR function serves the same purpose as the SUBSTRING function, but the syntax of the two functions differs.

To return a portion of a character string, specify the *start position* and *length* (optional) to determine which portion of the character string the SUBSTR function returns. The start position that you specify for the SUBSTR function can be a positive or a negative number. However, the SUBSTR function treats a negative number in the start position differently than does the SUBSTRING function. When the start position is a negative number, the SUBSTR function counts backward from the end of the character string, which depends on the length of the string, not the character length of a word or visible characters that the string contains. The SUBSTR function recognizes zero (`0`) or `1` in the start position as the first position in the string.

The following query shows an example of the SUBSTR function that includes a negative number for the start position. Given a start position of `-15`, the SUBSTR function counts backward 15 positions from the end of the string to find the start position and then returns the next five characters.

Figure 246. Query

```
SELECT sname, SUBSTR(sname, -15, 5) FROM state
   WHERE code = "CA";
```

Figure 247. Query result

```
sname          (expression)

California     Calif
```

To use a negative number for the start position, you need to know the length of the value that is evaluated. The **sname** column is defined as CHAR(15), so a SUBSTR function that accepts an argument of type **sname** can use a start position of `0`, `1`, or `-15` for the function to return a character string that begins from the first position in the string.

The following query returns the same result as Figure 246: Query on page 1608.

Figure 248. Query

```
SELECT sname, SUBSTR(sname, 1, 5) FROM state
   WHERE code = "CA";
```

The LPAD function

Use the LPAD function to return a copy of a string that has been left padded with a sequence of characters that are repeated as many times as necessary or truncated, depending on the specified length of the padded portion of the string. Specify the source string, the length of the string to be returned, and the character string to serve as padding.

The data type of the source string and the character string that serves as padding can be any data type that converts to VARCHAR or NVARCHAR.

The following query shows an example of the LPAD function with a specified length of 21 characters. Because the source string has a length of 15 characters (**sname** is defined as CHAR(15)), the LPAD function pads the first six positions to the left of the source string.

Figure 249. Query

```
SELECT sname, LPAD(sname, 21, "-")
   FROM state
   WHERE code = "CA" OR code = "AZ";
```

Figure 250. Query result

```
sname           (expression)

California      ------California
Arizona         ------Arizona
```

The RPAD function

Use the RPAD function to return a copy of a string that has been right padded with a sequence of characters that are repeated as many times as necessary or truncated, depending on the specified length of the padded portion of the string. Specify the source string, the length of the string to be returned, and the character string to serve as padding.

The data type of the source string and the character string that serves as padding can be any data type that converts to VARCHAR or NVARCHAR.

The following query shows an example of the RPAD function with a specified length of 21 characters. Because the source string has a length of 15 characters (**sname** is defined as CHAR(15)), the RPAD function pads the first six positions to the right of the source string.

Figure 251. Query

```
SELECT sname, RPAD(sname, 21, "-")
   FROM state
   WHERE code = "WV" OR code = "AZ";
```

Figure 252. Query result

```
sname          (expression)
West Virginia  West Virginia  ------
Arizona        Arizona        ------
```

In addition to these functions, the LTRIM and RTRIM functions can return a value that drops specified leading or trailing padding characters from their string argument, and the ASCII function can return the numeric value of the codepoint within the ASCII character set of the first character in its string argument. These built-in functions for operations on string values are described in the*HCL® Informix® Guide to SQL: Syntax*.

Other functions

You can also use the LENGTH, USER, CURRENT, SYSDATE, and TODAY functions anywhere in an SQL expression that you would use a constant. In addition, you can include the DBSERVERNAME function in a SELECT statement to display the name of the database server where the current database resides.

You can use these functions to select an expression that consists entirely of constant values or an expression that includes column data. In the first instance, the result is the same for all rows of output.

In addition, you can use the HEX function to return the hexadecimal encoding of an expression, the **ROUND** function to return the rounded value of an expression, and the **TRUNC** function to return the truncated value of an expression. For more information on the preceding functions, see the *HCL® Informix® Guide to SQL: Syntax*.

The LENGTH function

In the following query, the LENGTH function calculates the number of bytes in the combined **fname** and **lname** columns for each row where the length of **company** is greater than 15.

Figure 253. Query

```
SELECT customer_num,
   LENGTH (fname) + LENGTH (lname) namelength
   FROM customer
   WHERE LENGTH (company) > 15;
```

Figure 254. Query result

```
customer_num    namelength

         101            11
         105            13
         107            11
         112            14
         115            11
         118            10
         119            10
         120            10
         122            12
         124            11
         125            10
         126            12
         127            10
         128            11
```

Although the LENGTH function might not be useful when you work with , it can be important to determine the string length for programs and reports. The LENGTH function returns the clipped length of a CHARACTER or VARCHAR string and the full number of bytes in a TEXT or BYTE string.

HCL Informix® also supports the CHAR_LENGTH function, which returns the number of logical characters in its string argument, rather than the number of bytes. This function is useful in locales where a single logical character might require more than a single byte of storage. For more information about the CHAR_LENGTH function, see the *HCL® Informix® Guide to SQL: Syntax* and the *HCL® Informix® GLS User's Guide*.

The USER function

Use the USER function when you want to define a restricted view of a table that contains only rows that include your user ID. For information about how to create views, see the *IBM® Informix® Database Design and Implementation Guide* and the GRANT and CREATE VIEW statements in the *HCL® Informix® Guide to SQL: Syntax*.

The following query returns the user name (login account name) of the user who executes the query. It is repeated once for each row in the table.

Figure 255. Query

```
SELECT * FROM cust_calls
   WHERE user_id = USER;
```

If the user name of the current user is **richc**, the query retrieves only those rows in the **cust_calls** table where `user_id = richc`.

Figure 256. Query result

```
customer_num  110
call_dtime    1998-07-07 10:24
user_id       richc
call_code     L
call_descr    Order placed one month ago (6/7) not received.
res_dtime     1998-07-07 10:30
res_descr     Checked with shipping (Ed Smith). Order sent yesterday-we
              were waiting for goods from ANZ. Next time will call with
              delay if necessary

customer_num  119
call_dtime    1998-07-01 15:00
user_id       richc
call_code     B
call_descr    Bill does not reflect credit from previous order
res_dtime     1998-07-02 08:21
res_descr     Spoke with Jane Akant in Finance. They found the error and are
              sending new bill to customer
```

The TODAY function

The TODAY function returns the current system date. If the following query is issued when the current system date is July 10, 1998, it returns this one row.

Figure 257. Query

```
SELECT * FROM orders WHERE order_date = TODAY;
```

Figure 258. Query result

```
order_num      1018
order_date     07/10/1998
customer_num   121
ship_instruct  SW corner of Biltmore Mall
backlog        n
po_num         S22942
ship_date      07/13/1998
ship_weight    70.50
ship_charge    $20.00
paid_date      08/06/1998
```

The DBSERVERNAME and SITENAME functions

You can include the function DBSERVERNAME (or its synonym, SITENAME) in a SELECT statement to find the name of the database server. You can query the DBSERVERNAME for any table that has rows, including system catalog tables.

In the following query, you assign the label **server** to the DBSERVERNAME expression and also select the **tabid** column from the **systables** system catalog table. This table describes database tables, and **tabid** is the table identifier.

Figure 259. Query

```
SELECT DBSERVERNAME server, tabid
   FROM systables
   WHERE tabid <= 4;
```

Figure 260. Query result

```
 server           tabid

montague             1
montague             2
montague             3
montague             4
```

The WHERE clause restricts the numbers of rows displayed. Otherwise, the database server name would be displayed once for each row of the **systables** table.

The HEX function

In the following query, the HEX function returns the hexadecimal format of two columns in the **customer** table, as the result shows.

Figure 261. Query

```
SELECT HEX (customer_num) hexnum, HEX (zipcode) hexzip
   FROM customer;
```

Figure 262. Query result

```
hexnum     hexzip

0x00000065 0x00016F86
0x00000066 0x00016FA5
0x00000067 0x0001705F
0x00000068 0x00016F4A
0x00000069 0x00016F46
0x0000006A 0x00016F6F
⋮
```

The DBINFO function

You can call the DBINFO function in a SELECT statement to find any of the following information:

- The name of a dbspace corresponding to a tblspace number or expression
- The last SERIAL, SERIAL8 or BIGSERIAL value inserted into a table
- The number of rows processed by the SELECT, INSERT, DELETE, UPDATE, MERGE, EXECUTE FUNCTION, EXECUTE PROCEDURE, or EXECUTE ROUTINE statement
- The session ID of the current session
- The name of the current database to which the session is connected
- Whether an INSERT, UPDATE, or DELETE statement is being performed as part of a replicated transaction.
- The name of the host computer on which the database server runs
- The type of operating system and the word length of the host computer
- The local time zone and the current date and time in Coordinated Universal Time (UTC) format

- The DATETIME value corresponding to a specified integer column or to a specified UTC time value (as an integer number of seconds since 1970-01-01 00:00:00+00:00)
- The exact version of the database server to which a client application is connected, or a specified component of the full version string.

You can use the DBINFO function anywhere within SQL statements and within SPL routines.

The following query shows how you might use the DBINFO function to find out the name of the host computer on which the database server runs.

Figure 263. Query

```
SELECT FIRST 1 DBINFO('dbhostname') FROM systables;
```

Figure 264. Query result

```
(constant)

lyceum
```

Without the `FIRST 1` clause to restrict the values in the **tabid**, the host name of the computer on which the database server runs would be repeated for each row of the **systables** table. The following query shows how you might use the DBINFO function to find out the complete version number and the type of the current database server.

Figure 265. Query

```
SELECT FIRST 1 DBINFO('version','full') FROM systables;
```

For more information about how to use the DBINFO function to find information about your current database server, database session, or database, see the *HCL® Informix® Guide to SQL: Syntax*.

The DECODE function

You can use the DECODE function to convert an expression of one value to another value. The DECODE function has the following form:

```
DECODE(test, a, a_value, b, b_value, ..., n, n_value, exp_m )
```

The DECODE function returns *a_value* when *a* equals *test*, and returns *b_value* when *b* equals *test*, and, in general, returns *n_value* when *n* equals *test*.

If several expressions match *test*, DECODE returns *n_value* for the first expression found. If no expression matches *test*, DECODE returns *exp_m*; if no expression matches *test* and there is no *exp_m*, DECODE returns NULL.

Restriction: The DECODE function does not support arguments of type TEXT or BYTE.

Suppose an **employee** table exists that includes **emp_id** and **evaluation** columns. Suppose also that execution of the following query on the **employee** table returns the rows that the result shows.

Figure 266. Query

```
SELECT emp_id, evaluation FROM employee;
```

Figure 267. Query result

```
emp_id          evaluation

012233          great
012344          poor
012677          NULL
012288          good
012555          very good
```

In some cases, you might want to convert a set of values. For example, suppose you want to convert the descriptive values of the **evaluation** column in the preceding example to corresponding numeric values. The following query shows how you might use the DECODE function to convert values from the **evaluation** column to numeric values for each row in the **employee** table.

Figure 268. Query

```
SELECT emp_id, DECODE(evaluation, "poor", 0, "fair", 25, "good",
50, "very good", 75, "great", 100, -1) AS evaluation
   FROM employee;
```

Figure 269. Query result

```
emp_id         evaluation

012233          100
012344          0
012677          -1
012288          50
012555          75
```

You can specify any data type for the arguments of the DECODE function provided that the arguments meet the following requirements:

- The arguments *test*, *a*,*b*, ..., *n* all have the same data type or evaluate to a common compatible data type.
- The arguments *a_value*, *b_value*, ..., *n_value* all have the same data type or evaluate to a common compatible data type.

The NVL function

You can use the NVL function to convert an expression that evaluates to NULL to a value that you specify. The NVL function accepts two arguments: the first argument takes the name of the expression to be evaluated; the second argument specifies the value that the function returns when the first argument evaluates to NULL. If the first argument does not evaluate to NULL, the function returns the value of the first argument. Suppose a **student** table exists that includes **name** and **address** columns. Suppose also that execution of the following query on the **student** table returns the rows that the result shows.

Figure 270. Query

```
SELECT name, address FROM student;
```

Figure 271. Query result

```
name              address

John Smith        333 Vista Drive
Lauren Collier    1129 Greenridge Street
Fred Frith        NULL
Susan Jordan      NULL
```

The following query includes the NVL function, which returns a new value for each row in the table where the **address** column contains a NULL value.

Figure 272. Query

```
SELECT name, NVL(address, "address is unknown") AS address
   FROM student;
```

Figure 273. Query result

```
name              address

John Smith        333 Vista Drive
Lauren Collier    1129 Greenridge Street
Fred Frith        address is unknown
Susan Jordan      address is unknown
```

You can specify any data type for the arguments of the NVL function provided that the two arguments evaluate to a common compatible data type.

If both arguments of the NVL function evaluate to NULL, the function returns NULL.

HCL Informix® also supports the NULLIF function, which resembles the NVL function, but has different semantics. NULLIF returns NULL if its two arguments are equal, or returns its first argument if its arguments are not equal. For more information about the NULLIF function, see the *HCL® Informix® Guide to SQL: Syntax*.

SPL routines in SELECT statements

Previous examples in this chapter show SELECT statement expressions that consist of column names, operators, and SQL functions. This section shows expressions that contain an SPL routine call.

SPL routines contain special Stored Procedure Language (SPL) statements as well as SQL statements. For more information on SPL routines, see Create and use SPL routines on page 1753.

HCL Informix® allows you to write external routines in C and in Java™. For more information, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

When you include an SPL routine expression in a projection list, the SPL routine must be one that returns a single value (one column of one row). For example, the following statement is valid only if test_func() returns a single value:

```
SELECT col_a, test_func(col_b) FROM tab1
   WHERE col_c = "Davis";
```

SPL routines that return more than a single value are not supported in the Projection clause of SELECT statements. In the preceding example, if test_func() returns more than one value, the database server returns an error message.

SPL routines provide a way to extend the range of functions available by allowing you to perform a subquery on each row you select.

For example, suppose you want a listing of the customer number, the customer's last name, and the number of orders the customer has made. The following query shows one way to retrieve this information. The **customer** table has **customer_num** and **lname** columns but no record of the number of orders each customer has made. You could write a **get_orders** routine, which queries the **orders** table for each **customer_num** and returns the number of corresponding orders (labeled **n_orders**).
Figure 274. Query

```
SELECT customer_num, lname, get_orders(customer_num) n_orders
   FROM customer;
```

The result shows the output from this SPL routine.

Figure 275. Query result

```
customer_num    lname    n_orders

         101    Pauli           1
         102    Sadler          9
         103    Currie          9
         104    Higgins         4

         123    Hanlon          1
         124    Putnum          1
         125    Henry           0
         126    Neelie          1
         127    Satifer         1
         128    Lessor          0
```

Use SPL routines to encapsulate operations that you frequently perform in your queries. For example, the condition in the following query contains a routine, **conv_price**, that converts the unit price of a stock item to a different currency and adds any import tariffs.
Figure 276. Query

```
SELECT stock_num, manu_code, description FROM stock
   WHERE conv_price(unit_price, ex_rate = 1.50,
   tariff = 50.00) < 1000;
```

Data encryption functions

You can use the SET ENCRYPTION PASSWORD statement with built-in SQL encryption functions that use Advanced Encryption Standard (AES) and Triple DES (3DES) encryption to secure your sensitive data. When you use encryption, only those users who have the correct password will be able to read, copy, or modify the data.

Use the SET ENCRYPTION PASSWORD statement with the following built-in encryption and decryption functions:

- ENCRYPT_AES

```
ENCRYPT_AES(data-string-expression
[, password-string-expression [, hint-string-expression ]])
```

- ENCRYPT_TDES

```
ENCRYPT_TDES (data-string-expression
 [, password-string-expression [, hint-string-expression ]])
```

- DECRYPT_CHAR

```
DECRYPT_CHAR(EncryptedData [, PasswordOrPhrase])
```

- DECRYPT_BINARY

```
DECRYPT_BINARY(EncryptedData [, PasswordOrPhrase])
```

- GETHINT

```
GETHINT(EncryptedData)
```

If you have used the SET ENCRYPTION PASSWORD statement to specify a default password, then the database server applies that password in subsequent calls to encryption and decryption functions that you invoke in the same session.

Use ENCRYPT_AES and ENCRYPT_TDES to define encrypted data and use DECRYPT_CHAR and DECRYPT_BINARY to query encrypted data. Use GETHINT to display the password hint string, if set, on the server.

You can use these SQL built-in functions to implement column-level or cell-level encryption.

- Use column-level encryption to encrypt all values in a given column with the same password.
- Use cell-level encryption to encrypt data within the column with different passwords.



Tip: If you intend to select encrypted data from a large table, specify an unencrypted column on which to select the rows. You can create indexes or foreign-key constraints on columns that contain encrypted data, but to do so is an inefficient use of resources, because such indexes and foreign-key constraints are not used by the query optimizer.

Using column-level data encryption to secure credit card data

About this task

The following example uses column-level encryption to secure credit card data.

To use column-level data encryption to secure credit card data:

1. Create the table: `create table customer (id char(30), creditcard lvarchar(67));`
2. Insert the encryption data:
    a. Set session password: `SET ENCRYPTION PASSWORD "credit card number is encrypted";`
    b. Encrypt data.

```
    INSERT INTO customer VALUES
("Alice",  encrypt_aes("1234567890123456"));
    INSERT INTO customer VALUES
("Bob", encrypt_aes("2345678901234567"));
```

3. Query encryption data with decryption function.

```
SET ENCRYPTION PASSWORD "credit card number is encrypted";
SELECT id FROM customer
   WHERE DECRYPT_CHAR(creditcard) = "2345678901234567";
```

Results

Important: Encrypted data values occupy more storage space than the corresponding unencrypted data. A column whose width is sufficient to store plain text might need to be increased before it can support column-level encryption or cell-level encryption. If you attempt to insert an encrypted value into a column whose declared width is shorter than the encrypted string, the column stores a truncated value that cannot be decrypted.

For more information on encryption security, see *HCL® Informix® Administrator's Guide*.

For more information on the syntax and storage requirements of built-in encryption and decryption functions, see *HCL® Informix® Guide to SQL: Syntax*.

Summary

This chapter introduced sample syntax and results for functions in basic SELECT statements to query a relational database and to manipulate the returned data. Functions in SELECT statements on page 1592 shows how to perform the following actions:

- Use the aggregate functions in the Projection clause to calculate and retrieve specific data.
- Include the time functions DATE, DAY, MDY, MONTH, WEEKDAY, YEAR, CURRENT, and EXTEND plus the TODAY, LENGTH, and USER functions in your SELECT statements.
- Use conversion functions in the SELECT clause to convert between date and character values.
- Use string-manipulation functions in the SELECT clause to convert between upper and lower case letters or to manipulate character strings in various ways.

SPL routines in SELECT statements on page 1616 shows how to include SPL routines in your SELECT statements.

Data encryption functions on page 1617 shows how the use of the SET ENCRYPTION statement and built-in encryption and decryption functions can prevent users who cannot provide a password from viewing or modifying sensitive data.

Compose advanced SELECT statements

This section increases the scope of what you can do with the SELECT statement and enables you to perform more complex database queries and data manipulation. Compose SELECT statements on page 1532, focused on five of the clauses in the SELECT statement syntax. This section adds the GROUP BY clause and the HAVING clause. You can use the GROUP BY

clause with aggregate functions to organize rows returned by the FROM clause. You can include a HAVING clause to place conditions on the values that the GROUP BY clause returns.

This section also extends the earlier discussion of joins. It illustrates *self-joins*, which enable you to join a table to itself, and four kinds of *outer joins*, in which you apply the keyword OUTER to treat two or more joined tables unequally. It also introduces correlated and uncorrelated subqueries and their operational keywords, shows how to combine queries with the UNION operator, and defines the set operations known as union, intersection, and difference.

Examples in this section show how to use some or all of the SELECT statement clauses in your queries. The clauses must appear in the following order:

1. Projection
2. FROM
3. WHERE
4. GROUP BY
5. HAVING
6. ORDER BY
7. INTO TEMP

For an example of a SELECT statement that uses all these clauses in the correct order, see Figure 291: Query on page 1624.

An additional SELECT statement clause, INTO, which you can use to specify program and host variables in SQL APIs, is described in SQL programming on page 1700, as well as in the publications that come with the product.

This section also describes nested SELECT statements, in which subqueries are specified within the Projection, FROM, or WHERE clauses of the main query. Other sections show how SELECT statements can define and manipulate collections, and how to perform set operations on query results.

The GROUP BY and HAVING clauses

The optional GROUP BY and HAVING clauses add functionality to your SELECT statement. You can include one or both in a basic SELECT statement to increase your ability to manipulate aggregates.

The GROUP BY clause combines similar rows, producing a single result row for each group of rows that have the same values, for each column listed in the Projection clause. The HAVING clause sets conditions on those groups after you form them. You can use a GROUP BY clause without a HAVING clause, or a HAVING clause without a GROUP BY clause.

The GROUP BY clause

The GROUP BY clause divides a table into sets. This clause is most often combined with aggregate functions that produce summary values for each of those sets. Some examples in Compose SELECT statements on page 1532 show the use of aggregate functions applied to a whole table. This section illustrates aggregate functions applied to groups of rows.

Using the GROUP BY clause without aggregates is much like using the DISTINCT (or UNIQUE) keyword in the SELECT clause. The following query is described in Select specific columns on page 1542.

Figure 277. Query

```
SELECT DISTINCT customer_num FROM orders;
```

You could also write the statement as the following query shows.

Figure 278. Query

```
SELECT customer_num FROM orders
   GROUP BY customer_num;
```

Figure 277: Query on page 1621 and Figure 278: Query on page 1621 return the following rows.

Figure 279. Query result

```
customer_num

         101
         104
         106
         110

         124
         126
         127
```

The GROUP BY clause collects the rows into sets so that each row in each set has the same customer numbers. With no other columns selected, the result is a list of the unique **customer_num** values.

The power of the GROUP BY clause is more apparent when you use it with aggregate functions.

The following query retrieves the number of items and the total price of all items for each order.

Figure 280. Query

```
SELECT order_num, COUNT (*) number, SUM (total_price) price
   FROM items
   GROUP BY order_num;
```

The GROUP BY clause causes the rows of the **items** table to be collected into groups, each group composed of rows that have identical **order_num** values (that is, the items of each order are grouped together). After the database server forms the groups, the aggregate functions COUNT and SUM are applied within each group.

Figure 280: Query on page 1621 returns one row for each group. It uses labels to give names to the results of the COUNT and SUM expressions, as the result shows.

Figure 281. Query result

```
order_num      number         price

    1001           1          $250.00
    1002           2         $1200.00
    1003           3          $959.00
    1004           4         $1416.00

    1021           4         $1614.00
    1022           3          $232.00
    1023           6          $824.00
```

The result collects the rows of the **items** table into groups that have identical order numbers and computes the COUNT of rows in each group and the SUM of the prices.

You cannot include a TEXT, BYTE, CLOB, or BLOB column in a GROUP BY clause. To *group*, you must be able to *sort*, and no natural sort order exists for these data types.

Unlike the ORDER BY clause, the GROUP BY clause does not order data. Include an ORDER BY clause *after* your GROUP BY clause if you want to sort data in a particular order or sort on an aggregate in the projection list.

The following query is the same as Figure 280: Query on page 1621 but includes an ORDER BY clause to sort the retrieved rows in ascending order of **price**, as the result shows.

Figure 282. Query

```
SELECT order_num, COUNT(*) number, SUM (total_price) price
   FROM items
   GROUP BY order_num
   ORDER BY price;
```

Figure 283. Query result

```
 order_num      number         price

    1010           2          $84.00
    1011           1          $99.00
    1013           4         $143.80
    1022           3         $232.00
    1001           1         $250.00
    1020           2         $438.00
    1006           5         $448.00

    1002           2        $1200.00
    1004           4        $1416.00
    1014           2        $1440.00
    1019           1        $1499.97
    1021           4        $1614.00
    1007           5        $1696.00
```

The topic Select specific columns on page 1542 describes how to use an integer in an ORDER BY clause to indicate the position of a column in the projection list. You can also use an integer in a GROUP BY clause to indicate the position of column names or display labels in the GROUP BY list.

The following query returns the same rows as Figure 282: Query on page 1622 shows.

Figure 284. Query

```
SELECT order_num, COUNT(*) number, SUM (total_price) price
   FROM items
   GROUP BY 1
   ORDER BY 3;
```

When you build a query, all non-aggregate columns that are in the projection list in the Projection clause must also be included in the GROUP BY clause. A SELECT statement with a GROUP BY clause must return only one row per group. Columns that are listed after GROUP BY are certain to reflect only one distinct value within a group, and that value can be returned. However, a column not listed after GROUP BY might contain different values in the rows that are contained in the group.

The following query shows how to use the GROUP BY clause in a SELECT statement that joins tables.

Figure 285. Query

```
SELECT o.order_num, SUM (i.total_price)
   FROM orders o, items i
   WHERE o.order_date > '01/01/98'
      AND o.customer_num = 110
      AND o.order_num = i.order_num
   GROUP BY o.order_num;
```

The query joins the **orders** and **items** tables, assigns table aliases to them, and returns the rows.

Figure 286. Query result

```
  order_num            (sum)

       1008          $940.00
       1015          $450.00
```

The HAVING clause

To complement a GROUP BY clause, use a HAVING clause to apply one or more qualifying conditions to groups after they are formed. The effect of the HAVING clause on groups is similar to the way the WHERE clause qualifies individual rows. One advantage of using a HAVING clause is that you can include aggregates in the search condition, whereas you cannot include aggregates in the search condition of a WHERE clause.

Each HAVING condition compares one column or aggregate expression of the group with another aggregate expression of the group or with a constant. You can use HAVING to place conditions on both column values and aggregate values in the group list.

The following query returns the average total price per item on all orders that have more than two items. The HAVING clause tests each group as it is formed and selects those that are composed of more than two rows.

Figure 287. Query

```
SELECT order_num, COUNT(*) number, AVG (total_price) average
   FROM items
   GROUP BY order_num
   HAVING COUNT(*) > 2;
```

Figure 288. Query result

```
  order_num      number          average

       1003           3          $319.67
       1004           4          $354.00
       1005           4          $140.50
       1006           5           $89.60
       1007           5          $339.20
       1013           4           $35.95
       1016           4          $163.50
       1017           3          $194.67
       1018           5          $226.20
       1021           4          $403.50
       1022           3           $77.33
       1023           6          $137.33
```

If you use a HAVING clause without a GROUP BY clause, the HAVING condition applies to all rows that satisfy the search condition. In other words, all rows that satisfy the search condition make up a single group.

The following query, a modified version of Figure 287: Query on page 1623, returns just one row, the average of all **total_price** values in the table, as the result shows.

Figure 289. Query

```
SELECT AVG (total_price) average
   FROM items
   HAVING count(*) > 2;
```

Figure 290. Query result

```
         average

         $270.97
```

If Figure 289: Query on page 1624, like Figure 287: Query on page 1623, had included the non-aggregate column **order_num** in the Projection clause, you would have to include a GROUP BY clause with that column in the group list. In addition, if the condition in the HAVING clause was not satisfied, the output would show the column heading and a message would indicate that no rows were found.

The following query contains all the SELECT statement clauses that you can use in the HCL® Informix® version of interactive SQL (the INTO clause that names host variables is available only in an SQL API).

Figure 291. Query

```
SELECT o.order_num, SUM (i.total_price) price,
     paid_date - order_date span
   FROM orders o, items i
   WHERE o.order_date > '01/01/98'
      AND o.customer_num > 110
      AND o.order_num = i.order_num
   GROUP BY 1, 3
   HAVING COUNT (*) < 5
   ORDER BY 3
   INTO TEMP temptab1;
```

The query joins the **orders** and **items** tables; employs display labels, table aliases, and integers that are used as column indicators; groups and orders the data; and puts the results in a temporary table, as the result shows.

Figure 292. Query result

```
  order_num            price        span

       1017          $584.00
       1016          $654.00
       1012         $1040.00
       1019         $1499.97          26
       1005          $562.00          28
       1021         $1614.00          30
       1022          $232.00          40
       1010           $84.00          66
       1009          $450.00          68
       1020          $438.00          71
```

Create advanced joins

The topic Create a join on page 1570 shows how to include a WHERE clause in a SELECT statement to join two or more tables on one or more columns. It illustrates natural joins and equi-joins.

This section discusses how to use two more complex kinds of joins, self-joins and outer joins. As described for simple joins, you can define aliases for tables and assign display labels to expressions to shorten your multiple-table queries. You can also issue a SELECT statement with an ORDER BY clause that sorts data into a temporary table.

Self-joins

A join does not always have to involve two different tables. You can join a table to itself, creating a *self-join*. Joining a table to itself can be useful when you want to compare values in a column to other values in the same column.

To create a self-join, list a table twice in the FROM clause, and assign it a different alias each time. Use the aliases to refer to the table in the Projection and WHERE clauses as if it were two separate tables. (Aliases in SELECT statements are discussed in Aliases on page 1577 and in the *HCL® Informix® Guide to SQL: Syntax*.)

Just as in joins between tables, you can use arithmetic expressions in self-joins. You can test for null values, and you can use an ORDER BY clause to sort the values in a specified column in ascending or descending order.

The following query finds pairs of orders where the **ship_weight** differs by a factor of five or more and the **ship_date** is not null. The query then orders the data by **ship_date**.

Figure 293. Query

```
SELECT x.order_num, x.ship_weight, x.ship_date,
       y.order_num, y.ship_weight, y.ship_date
   FROM orders x, orders y
   WHERE x.ship_weight >= 5 * y.ship_weight
      AND x.ship_date IS NOT NULL
      AND y.ship_date IS NOT NULL
   ORDER BY x.ship_date;
```

Table 85. Query result

order_num	ship_weight	ship_date	order_num	ship_weight	ship_date
1004	95.80	05/30/1998	1011	10.40	07/03/1998
1004	95.80	05/30/1998	1020	14.00	07/16/1998
1004	95.80	05/30/1998	1022	15.00	07/30/1998
1007	125.90	06/05/1998	1015	20.60	07/16/1998
1007	125.90	06/05/1998	1020	14.00	07/16/1998

If you want to store the results of a self-join into a temporary table, append an INTO TEMP clause to the SELECT statement and assign display labels to at least one set of columns to rename them. Otherwise, the duplicate column names cause an error and the temporary table is not created.

The following query, which is similar to Figure 293: Query on page 1625, labels all columns selected from the **orders** table and puts them in a temporary table called **shipping**.

Figure 294. Query

```
SELECT x.order_num orders1, x.po_num purch1,
       x.ship_date ship1, y.order_num orders2,
       y.po_num purch2, y.ship_date ship2
   FROM orders x, orders y
   WHERE x.ship_weight >= 5 * y.ship_weight
      AND x.ship_date IS NOT NULL
      AND y.ship_date IS NOT NULL
   ORDER BY orders1, orders2
   INTO TEMP shipping;
```

If you query with `SELECT *` from table **shipping**, you see the following rows.

Figure 295. Query result

```
    orders1 purch1     ship1          orders2 purch2    ship2

       1004 8006       05/30/1998        1011 B77897    07/03/1998
       1004 8006       05/30/1998        1020 W2286     07/16/1998
       1004 8006       05/30/1998        1022 W9925     07/30/1998
       1005 2865       06/09/1998        1011 B77897    07/03/1998

       1019 Z55709     07/16/1998        1020 W2286     07/16/1998
       1019 Z55709     07/16/1998        1022 W9925     07/30/1998
       1023 KF2961     07/30/1998        1011 B77897    07/03/1998
```

You can join a table to itself more than once. The maximum number of self-joins depends on the resources available to you.

The self-join in the following query creates a list of those items in the **stock** table that are supplied by three manufacturers. The self-join includes the last two conditions in the WHERE clause to eliminate duplicate manufacturer codes in rows that are retrieved.

Figure 296. Query

```
SELECT s1.manu_code, s2.manu_code, s3.manu_code,
       s1.stock_num, s1.description
   FROM stock s1, stock s2, stock s3
   WHERE s1.stock_num = s2.stock_num
      AND s2.stock_num = s3.stock_num
      AND s1.manu_code < s2.manu_code
      AND s2.manu_code < s3.manu_code
   ORDER BY stock_num;
```

Figure 297. Query result

```
manu_code manu_code manu_code stock_num description

HRO       HSK       SMT               1 baseball gloves
ANZ       NRG       SMT               5 tennis racquet
ANZ       HRO       HSK             110 helmet
ANZ       HRO       PRC             110 helmet
ANZ       HRO       SHM             110 helmet
ANZ       HSK       PRC             110 helmet
ANZ       HSK       SHM             110 helmet
ANZ       PRC       SHM             110 helmet
HRO       HSK       PRC             110 helmet
HRO       HSK       SHM             110 helmet
HRO       PRC       SHM             110 helmet

KAR       NKL       PRC             301 running shoes
KAR       NKL       SHM             301 running shoes
KAR       PRC       SHM             301 running shoes
NKL       PRC       SHM             301 running shoes
```

If you want to select rows from a payroll table to determine which employees earn more than their manager, you might construct the self-join as the following SELECT statement shows:

```
SELECT emp.employee_num, emp.gross_pay, emp.level,
       emp.dept_num, mgr.employee_num, mgr.gross_pay,
       mgr.dept_num, mgr.level
   FROM payroll emp, payroll mgr
   WHERE emp.gross_pay > mgr.gross_pay
      AND emp.level < mgr.level
      AND emp.dept_num = mgr.dept_num
   ORDER BY 4;
```

The following query uses a *correlated subquery* to retrieve and list the 10 highest-priced items ordered.

Figure 298. Query

```
SELECT order_num, total_price
   FROM items a
   WHERE 10 >
      (SELECT COUNT (*)
         FROM items b
         WHERE b.total_price < a.total_price)
   ORDER BY total_price;
```

The query returns the 10 rows.

Figure 299. Query result

```
order_num    total_price

   1018      $15.00
   1013      $19.80
   1003      $20.00
   1005      $36.00
   1006      $36.00
   1013      $36.00
   1010      $36.00
   1013      $40.00
   1022      $40.00
   1023      $40.00
```

You can create a similar query to find and list the 10 employees in the company who have the most seniority.

For more information about correlated subqueries, refer to Subqueries in SELECT statements on page 1635.

Outer joins

This section shows how to create and use outer joins in a SELECT statement. The topic Create a join on page 1570 discusses inner joins. Whereas an inner join treats two or more joined tables equally, an outer join treats two or more joined tables asymmetrically. An outer join makes one of the tables dominant (also called the *outer table*) over the other subordinate tables (also called *inner tables*).

In an inner join or in a *simple join*, the result contains only the combinations of rows that satisfy the join conditions. Rows that do not satisfy the join conditions are discarded.

In an outer join, the result contains the combinations of rows that satisfy the join conditions and the rows from the dominant table that would otherwise be discarded because no matching row was found in the subordinate table. The rows from the dominant table that do not have matching rows in the subordinate table contain NULL values in the columns selected from the subordinate table.

An outer join allows you to apply *join filters* to the inner table before the join condition is applied.

The database server supports syntax for outer joins that is an extension to the ANSI standard for the SQL language. Besides outer join syntax based on this HCL® Informix® extension, the database server also supports the ANSI standard syntax, which provides more flexibility for creating queries that join a dominant table with one or more subordinate tables. With the exception of joins in view definitions, it is recommended that you use the ANSI standard syntax for creating new outer-join queries.

In view definitions, however, the Informix®-extension syntax does not require materialized views, and so it might offer performance advantages over ISO/ANSI join syntax in those contexts, including business-analytic operations that query complex views joining multiple tables.

Whichever form of outer-join syntax you use, however, a single query cannot mix both syntax modes. All of the outer join operations in the same query block must either use SQL syntax that complies with the ISO/ANSI standard, or else use the HCL® Informix® extension syntax.

Important: Before you rely on outer joins, determine whether one or more inner joins can work. You can often use an inner join when you do not need supplemental information from other tables.

Restriction: You cannot combine HCL® Informix® and ANSI outer-join syntax in the same query block.

For information on the syntax of outer joins, see the *HCL® Informix® Guide to SQL: Syntax*.

HCL Informix® extension to outer join syntax

The HCL Informix® extension to outer-join syntax begins an outer join with the OUTER keyword. When you use the Informix® syntax, you must include the join condition in the WHERE clause. When you use the Informix® syntax for an outer join, the database server supports the following three basic types of outer joins:

- A simple outer join on two tables
- An outer join for a simple join to a third table
- An outer join of two tables to a third table

An outer join must have a Projection clause, a FROM clause, and a WHERE clause. The join conditions are expressed in a WHERE clause. To transform a simple join into an outer join, insert the keyword OUTER directly before the name of the subordinate tables in the FROM clause. As shown later in this section, you can include the OUTER keyword more than once in your query.

No Informix® extension to outer-join syntax is equivalent to the ANSI right outer join.

ANSI join syntax

The following ANSI joins are supported:

- Left outer join
- Right outer join

The ANSI outer-join syntax begins an outer join with the LEFT JOIN, LEFT OUTER JOIN, RIGHT JOIN, or RIGHT OUTER JOIN keywords. The OUTER keyword is optional. Queries can specify a join condition and optional join filters in the ON clause. The WHERE clause specifies a post-join filter. In addition, you can explicitly specify the type of join using the LEFT or right clause. ANSI join syntax also allows the dominant or subordinate part of an outer join to be the result set of another join, when you begin the join with a left parenthesis.

If you use ANSI syntax for an outer join, you must use the ANSI syntax for all outer joins in a single query block.

Tip: The examples in this section use table aliases for brevity. Aliases on page 1577 discusses table aliases.

Left outer join

In the syntax of a left outer join, the dominant table of the outer join appears to the left of the keyword that begins the outer join. A left outer join returns all of the rows for which the join condition is true and, in addition, returns all other rows from the dominant table and displays the corresponding values from the subservient table as NULL.

The following query uses ANSI syntax for a LEFT OUTER JOIN to join the **customer** and **cust_calls** tables, with **customer** the dominant table:

Figure 300. Query 1

```
SELECT c.customer_num, c.lname, c.company, c.phone,
       u.call_dtime, u.call_descr
   FROM customer c LEFT OUTER JOIN cust_calls u
   ON c.customer_num = u.customer_num;
```

The next query similarly uses the ON clause to specify the join condition, and adds an additional filter in the WHERE clause to limit your result set; such a filter is a post-join filter.

The query returns only rows in which customers have not made any calls to customer service. In this query, the database server applies the filter in the WHERE clause after it performs the outer join on the **customer_num** column of the **customer** and **cust_calls** tables.

Figure 301. Query 2

```
SELECT c.customer_num, c.lname, c.company, c.phone,
   u.call_dtime, u.call_descr
   FROM customer c LEFT OUTER JOIN cust_calls u
   ON c.customer_num = u.customer_num
   WHERE u.customer_num IS NULL;
```

The next example shows the HCL® Informix®-extension syntax that us equivalent to the previous ANSI-compliant left outer join:

Figure 302. Query 3

```
SELECT c.customer_num, c.lname, c.company, c.phone,
   u.call_dtime, u.call_descr
   FROM customer c OUTER cust_calls u
   WHERE c.customer_num = u.customer_num
   AND u.customer_num IS NULL;
```

Here the WHERE clause defines the join condition, and excludes rows with non-NULL **cust_calls** values.

Examples of ISO/ANSI LEFT OUTER JOIN queries

The following examples illustrate various query constructions that ANSI join syntax can support:

```
SELECT *
FROM (t1 LEFT OUTER JOIN (t2 LEFT OUTER JOIN t3 ON t2.c1=t3.c1)
   ON t1.c1=t3.c1) JOIN (t4 LEFT OUTER JOIN t5 ON t4.c1=t5.c1)
   ON t1.c1=t4.c1;
```

```
SELECT *
FROM (t1 LEFT OUTER JOIN (t2 LEFT OUTER JOIN t3 ON t2.c1=t3.c1)
   ON t1.c1=t3.c1),
   (t4 LEFT OUTER JOIN t5 ON t4.c1=t5.c1)
   WHERE t1.c1 = t4.c1;

SELECT *
FROM (t1 LEFT OUTER JOIN (t2 LEFT OUTER JOIN t3 ON t2.c1=t3.c1)
   ON t1.c1=t3.c1) LEFT OUTER JOIN (t4 JOIN t5 ON t4.c1=t5.c1)
   ON t1.c1=t4.c1;

SELECT *
FROM t1 LEFT OUTER JOIN (t2 LEFT OUTER JOIN t3 ON t2.c1=t3.c1)
   ON t1.c1=t2.c1;

SELECT *
FROM t1 LEFT OUTER JOIN (t2 LEFT OUTER JOIN t3 ON t2.c1=t3.c1)
   ON t1.c1=t3.c1;

SELECT *
FROM (t1 LEFT OUTER JOIN t2 ON t1.c1=t2.c1)
   LEFT OUTER JOIN t3 ON t2.c1=t3.c1;

SELECT *
FROM (t1 LEFT OUTER JOIN t2 ON t1.c1=t2.c1)
   LEFT OUTER JOIN t3 ON t1.c1=t3.c1;

SELECT *
FROM t9, (t1 LEFT JOIN t2 ON t1.c1=t2.c1),
   (t3 LEFT JOIN t4 ON t3.c1=10), t10, t11,
   (t12 LEFT JOIN t14 ON t12.c1=100);

SELECT * FROM
  ((SELECT c1,c2 FROM t3) AS vt3(v31,v32)
    LEFT OUTER JOIN
        ( (SELECT c1,c2 FROM t1) AS vt1(vc1,vc2)
        LEFT OUTER JOIN
        (SELECT c1,c2 FROM t2) AS vt2(vc3,vc4)
         ON vt1.vc1 = vt2.vc3)
ON vt3.v31 = vt2.vc3);
```

The last example above includes ANSI-compliant joins on derived tables. It specifies a left outer join on the results of a subquery in the FROM clause of the outer query with the results of another left outer join on two other subquery results. See the section Subqueries in the FROM clause on page 1638 for less complex examples of the ANSI-compliant syntax for subqueries.

Right outer join

In the syntax of a right outer join, the dominant table of the outer join appears to the right of the keyword that begins the outer join. A right outer join returns all of the rows for which the join condition is true and, in addition, returns all other rows from the dominant table and displays the corresponding values from the subservient table as NULL.

The following query is an example of a right outer join on the **customer** and **orders** tables.

Figure 303. Query

```
SELECT c.customer_num, c.fname, c.lname, o.order_num,
o.order_date, o.customer_num
FROM customer c RIGHT OUTER JOIN orders o
ON (c.customer_num = o.customer_num);
```

The query returns all rows from the dominant table **orders** and, as necessary, displays the corresponding values from the subservient table **customer** as NULL.

Figure 304. Query result

```
customer_num   fname    lname order_num   order_date  customer_num
        104  Anthony  Wiggins      1001   05/30/1998           104
        101  Ludwig     Pauli      1002   05/30/1998           101
        104 Anthony   Wiggins      1003   05/30/1998           104
     <NULL>  <NULL>    <NULL>      1004   06/05/1998           106
```

Simple join

The following query is an example of a simple join on the **customer** and **cust_calls** tables.

Figure 305. Query

```
SELECT c.customer_num, c.lname, c.company,
     c.phone, u.call_dtime, u.call_descr
   FROM customer c, cust_calls u
   WHERE c.customer_num = u.customer_num;
```

The query returns only those rows in which the customer has made a call to customer service, as the result shows.

Figure 306. Query result

```
customer_num  106
lname         Watson
company       Watson & Son
phone         415-389-8789
call_dtime    1998-06-12 08:20
call_descr    Order was received, but two of the cans of
              ANZ tennis balls within the case were empty

customer_num  116
lname         Parmelee
company       Olympic City
phone         415-534-8822
call_dtime    1997-12-21 11:24
call_descr    Second complaint from this customer! Received
              two cases right-handed outfielder gloves (1 HRO)
              instead of one case lefties.
```

Simple outer join on two tables

The following query uses the same Projection clause, tables, and comparison condition as the preceding example, but this time it creates a simple outer join in HCL® Informix® extension syntax.

Figure 307. Query

```
SELECT c.customer_num, c.lname, c.company,
     c.phone, u.call_dtime, u.call_descr
   FROM customer c, OUTER cust_calls u
   WHERE c.customer_num = u.customer_num;
```

The addition of the keyword `OUTER` before the **cust_calls** table makes it the subservient table. An outer join causes the query to return information on all customers, whether or not they have made calls to customer service. All rows from the dominant **customer** table are retrieved, and NULL values are assigned to columns of the subservient **cust_calls** table, as the result shows.

Figure 308. Query result

```
customer_num  101
lname         Pauli
company       All Sports Supplies
phone         408-789-8075
call_dtime
call_descr

customer_num  102
lname         Sadler
company       Sports Spot
phone         415-822-1289
call_dtime
call_descr

customer_num  107
lname         Ream
company       Athletic Supplies
phone         415-356-9876
call_dtime
call_descr

customer_num  108
lname         Quinn
company       Quinn's Sports
phone         415-544-8729
call_dtime
call_descr
```

Outer join for a simple join to a third table

Using the HCL® Informix® syntax, the following query shows an outer join that is the result of a simple join to a third table. This second type of outer join is known as a *nested simple join*.

Figure 309. Query

```
SELECT c.customer_num, c.lname, o.order_num,
     i.stock_num, i.manu_code, i.quantity
   FROM customer c, OUTER (orders o, items i)
   WHERE c.customer_num = o.customer_num
      AND o.order_num = i.order_num
      AND manu_code IN ('KAR', 'SHM')
   ORDER BY lname;
```

The query first performs a simple join on the **orders** and **items** tables, retrieving information on all orders for items with a **manu_code** of KAR or SHM. It then performs an outer join to combine this information with data from the dominant **customer** table. An optional ORDER BY clause reorganizes the data into the following form.

Figure 310. Query result

```
customer_num lname            order_num stock_num manu_code quantity

         114 Albertson
         118 Baxter
         113 Beatty

         105 Vector
         121 Wallack               1018       302 KAR             3
         106 Watson
```

Outer join of two tables to a third table

Using the HCL® Informix® extension syntax, the following query shows an outer join that is the result of an outer join of each of two tables to a third table. In this third type of outer join, join relationships are possible only between the dominant table and the subservient tables.

Figure 311. Query

```
SELECT c.customer_num, c.lname, o.order_num,
     order_date, call_dtime
   FROM customer c, OUTER orders o, OUTER cust_calls x
   WHERE c.customer_num = o.customer_num
      AND c.customer_num = x.customer_num
   ORDER BY lname
   INTO TEMP service;
```

The query individually joins the subservient tables **orders** and **cust_calls** to the dominant **customer** table; it does not join the two subservient tables. An INTO TEMP clause selects the results into a temporary table for further manipulation or queries, as the result shows.

Figure 312. Query result

```
customer_num lname            order_num order_date call_dtime

         114 Albertson
         118 Baxter
         113 Beatty
         103 Currie
         115 Grant                 1010 06/17/1998

         117 Sipes                 1012 06/18/1998
         105 Vector
         121 Wallack               1018 07/10/1998 1998-07-10 14:05
         106 Watson                1004 05/22/1998 1998-06-12 08:20
         106 Watson                1014 06/25/1998 1998-06-12 08:20
```

If Figure 311: Query on page 1634 had tried to create a join condition between the two subservient tables **o** and **x**, as the following query shows, an error message would indicate the creation of a two-sided outer join.

Figure 313. Query

```
WHERE o.customer_num = x.customer_num
```

Joins that combine outer joins

To achieve multiple levels of nesting, you can create a join that employs any combination of the three types of outer joins. Using the ANSI syntax, the following query creates a join that is the result of a combination of a simple outer join on two tables and a second outer join.

Figure 314. Query

```
SELECT c.customer_num, c.lname, o.order_num,
     stock_num, manu_code, quantity
   FROM customer c, OUTER (orders o, OUTER items i)
   WHERE c.customer_num = o.customer_num
      AND o.order_num = i.order_num
      AND manu_code IN ('KAR', 'SHM')
   ORDER BY lname;
```

The query first performs an outer join on the **orders** and **items** tables, retrieving information on all orders for items with a **manu_code** of KAR or SHM. It then performs a second outer join that combines this information with data from the dominant **customer** table.

Figure 315. Query result

```
customer_num lname            order_num stock_num manu_code quantity

         114 Albertson
         118 Baxter
         113 Beatty
         103 Currie
         115 Grant                 1010

         117 Sipes                 1012
         117 Sipes                 1007
         105 Vector
         121 Wallack               1018       302 KAR             3
         106 Watson                1014
         106 Watson                1004
```

You can specify the join conditions in two ways when you apply an outer join to the result of an outer join to a third table. The two subservient tables are joined, but you can join the dominant table to either subservient table without affecting the results if the dominant table and the subservient table share a common column.

Subqueries in SELECT statements

A *subquery* (the inner SELECT statement, where one SELECT statement is nested within another) can return zero or more rows or expressions. Each subquery must be delimited by parentheses, and must contain a Projection clause and a FROM clause. A subquery can itself contain other subqueries.

The database server supports subqueries in the following contexts:

- A SELECT statement nested in the Projection clause of another SELECT statement
- a SELECT statement nested in the WHERE clause of another SELECT statement
- a SELECT statement nested in the FROM clause of another SELECT statement.

You can also specify a subquery in various clauses of the INSERT, DELETE, MERGE, or UPDATE statements where a subquery is valid.

Subqueries in the Projection clause or in the WHERE clause can be *correlated* or *uncorrelated*. A subquery is correlated when the value that it produces depends on a value produced by the outer SELECT statement that contains it. For more information, see Correlated subqueries on page 1636.

Any other kind of subquery is considered uncorrelated. Only uncorrelated subqueries are valid in the FROM clause of the SELECT statement.

Correlated subqueries

A *correlated subquery* is a subquery that refers to a column of a table that is not listed in its FROM clause. The column can be in the Projection clause or in the WHERE clause. To find the table to which the correlated subquery refers, search the columns until a correlation is found.

In general, correlated subqueries diminish performance. Use the table name or alias in the subquery so that there is no doubt as to which table the column is in.

The database server will use the outer query to get values. For example, if the table **taba** has the column **col1** and table **tabb** has the column **col2** and they contain the following:

```
taba.col1       aa,bb,null
tabb.col2       bb, null
```

And the query is:

```
select * from taba where col1 in (select col1 from tabb);
```

Then the results might be meaningless. The database server will provide all values in **taba.col1** and then compare them to **taba.col1** (outer query WHERE clause). This will return all rows. You usually use the subquery to return column values from the inner table. Had the query been written as:

```
select * from taba where col1 in (select tabb.col1 from tabb);
```

Then the `error -217 column not found` would have resulted.

The important feature of a correlated subquery is that, because it depends on a value from the outer SELECT, it must be executed repeatedly, once for every value that the outer SELECT produces. An uncorrelated subquery is executed only once.

Using subqueries to combine SELECT statements

You can construct a SELECT statement with a subquery to replace two separate SELECT statements.

Subqueries in SELECT statements allow you to perform various tasks, including the following actions:

- Compare an expression to the result of another SELECT statement
- Determine whether the results of another SELECT statement include a specific expression
- Determine whether another SELECT statement returns any rows

An optional WHERE clause in a subquery is often used to narrow the search condition.

A subquery selects and returns values to the first or outer SELECT statement. A subquery can return no value, a single value, or a set of values, as follows:

- If a subquery returns no value, the query does not return any rows. Such a subquery is equivalent to a NULL value.
- If a subquery returns one value, the value is in the form of either one aggregate expression or exactly one row and one column. Such a subquery is equivalent to a single number or character value.
- If a subquery returns a list or set of values, the values can represent one row or one column.
- In the FROM clause of the outer query, a subquery can represent a set of rows (sometimes called a *derived table* or a *table expression*).

Subqueries in a Projection clause

A subquery can occur in the Projection clause of another SELECT statement. The following query shows how you might use a subquery in a Projection clause to return the total shipping charges (from the **orders** table) for each customer in the **customer** table. You could also write this query as a join between two tables.

Figure 316. Query

```
SELECT customer.customer_num,
   (SELECT SUM(ship_charge)
       FROM orders
       WHERE customer.customer_num = orders.customer_num)
         AS total_ship_chg
   FROM customer;
```

Figure 317. Query result

```
customer_num total_ship_chg

         101         $15.30
         102
         103
         104         $38.00
         105
        ⋮
         123          $8.50
         124         $12.00
         125
         126         $13.00
         127         $18.00
         128
```

Subqueries in the FROM clause

This topic describes subqueries that occur as nested SELECT statements in the FROM clause of an outer SELECT statement. Such subqueries are sometimes called *derived tables* or *table expressions* because the outer query uses the results of the subquery as a data source.

The following query uses asterisk notation in the outer query to return the results of a subquery that retrieves all fields of the **address** column in the **employee** table.

Figure 318. Query

```
SELECT * FROM (SELECT address.* FROM employee);
```

Figure 319. Query result

```
address   ROW(102 Ruby, Belmont, CA, 49932, 1000)
address   ROW(133 First, San Jose, CA, 85744, 4900)
address   ROW(152 Topaz, Willits, CA, 69445, 1000))
```

This illustrates how to specify a derived table, but it is a trivial example of this syntax, because the outer query does not manipulate any values in the table expression that the subquery in the FROM clause returns. (See Figure 158: Query on page 1585 for a simple query that returns the same results.)

The following query is a more complex example in which the outer query selects only the first qualifying row of a derived table that a subquery in the FROM clause specifies as a simple join on the **customer** and **cust_calls** tables.

Figure 320. Query

```
SELECT LIMIT 1 * FROM
   (SELECT c.customer_num, c.lname, c.company,
           c.phone, u.call_dtime, u.call_descr
                 FROM customer c, cust_calls u
                 WHERE c.customer_num = u.customer_num
    ORDER BY u.call_dtime DESC);
```

The query returns only those rows in which the customer has made a call to customer service, as the result shows.

Figure 321. Query result

```
customer_num  106
lname         Watson
company       Watson & Son
phone         415-389-8789
call_dtime    1998-06-12 08:20
call_descr    Order was received, but two of the cans of
              ANZ tennis balls within the case were empty
```

In the preceding example, the subquery includes an ORDER BY clause that specifies a column that appears in Projection list of the subquery, but the query would also be valid if the Projection list had omitted the **u.call_dtime** column. The FROM clause is the only context in which a subquery can specify the ORDER BY clause.

Subqueries in WHERE clauses

This section describes subqueries that occur as a SELECT statement that is nested in the WHERE clause of another SELECT statement.

You can use any relational operator with ALL and ANY to compare something to every one of (ALL) or to any one of (ANY) the values that the subquery produces. You can use the keyword SOME in place of ANY. The operator IN is equivalent to = ANY. To create the opposite search condition, use the keyword NOT or a different relational operator.

The EXISTS operator tests a subquery to see if it found any values; that is, it asks if the result of the subquery is not null. You cannot use the EXISTS keyword in a subquery that contains a column with a TEXT or BYTE data type.

For the syntax that you use to create a condition with a subquery, see the *HCL® Informix® Guide to SQL: Syntax*.

The following keywords introduce a subquery in the WHERE clause of a SELECT statement.

The ALL keyword

Use the keyword ALL preceding a subquery to determine whether a comparison is true for every value returned. If the subquery returns no values, the search condition is *true*. (If it returns no values, the condition is true of all the zero values.)

The following query lists the following information for all orders that contain an item for which the total price is less than the total price on every item in order number 1023.

Figure 322. Query

```
SELECT order_num, stock_num, manu_code, total_price
   FROM items
   WHERE total_price < ALL
      (SELECT total_price FROM items
         WHERE order_num = 1023);
```

Figure 323. Query result

```
  order_num stock_num manu_code total_price

       1003         9 ANZ            $20.00
       1005         6 SMT            $36.00
       1006         6 SMT            $36.00
       1010         6 SMT            $36.00
       1013         5 ANZ            $19.80
       1013         6 SMT            $36.00
       1018       302 KAR            $15.00
```

The ANY keyword

Use the keyword ANY (or its synonym SOME) before a subquery to determine whether a comparison is true for at least one of the values returned. If the subquery returns no values, the search condition is *false*. (Because no values exist, the condition cannot be true for one of them.)

The following query finds the order number of all orders that contain an item for which the total price is greater than the total price of any one of the items in order number 1005.

Figure 324. Query

```
SELECT DISTINCT order_num
   FROM items
   WHERE total_price > ANY
      (SELECT total_price
         FROM items
         WHERE order_num = 1005);
```

Figure 325. Query result

```
order_num

       1001
       1002
       1003
       1004

       1020
       1021
       1022
       1023
```

Single-valued subqueries

You do not need to include the keyword ALL or ANY if you know the subquery can return exactly one value to the outer-level query. A subquery that returns exactly one value can be treated like a function. This kind of subquery often uses an aggregate function because aggregate functions always return single values.

The following query uses the aggregate function MAX in a subquery to find the **order_num** for orders that include the maximum number of volleyball nets.

Figure 326. Query

```
SELECT order_num FROM items
   WHERE stock_num = 9
      AND quantity =
         (SELECT MAX (quantity)
            FROM items
            WHERE stock_num = 9);
```

Figure 327. Query result

```
  order_num

       1012
```

The following query uses the aggregate function MIN in the subquery to select items for which the total price is higher than 10 times the minimum price.

Figure 328. Query

```
SELECT order_num, stock_num, manu_code, total_price
   FROM items x
   WHERE total_price >
      (SELECT 10 * MIN (total_price)
         FROM items
         WHERE order_num = x.order_num);
```

Figure 329. Query result

```
order_num stock_num manu_code  total_price

     1003         8 ANZ           $840.00
     1018       307 PRC           $500.00
     1018       110 PRC           $236.00
     1018       304 HRO           $280.00
```

Correlated subqueries

A *correlated subquery* is a subquery that refers to a column of a table that is not in its FROM clause. The column can be in the Projection clause or in the WHERE clause.

In general, correlated subqueries diminish performance. It is recommended that you qualify the column name in subqueries with the name or alias of the table, in order to remove any doubt regarding in which table the column resides.

The following query is an example of a correlated subquery that returns a list of the 10 latest shipping dates in the **orders** table. It includes an ORDER BY clause after the subquery to order the results because (except in the FROM clause) you cannot include ORDER BY within a subquery.

Figure 330. Query

```
SELECT po_num, ship_date FROM orders main
   WHERE 10 >
      (SELECT COUNT (DISTINCT ship_date)
         FROM orders sub
         WHERE sub.ship_date < main.ship_date)
   AND ship_date IS NOT NULL
   ORDER BY ship_date, po_num;
```

The subquery is correlated because the number that it produces depends on **main.ship_date**, a value that the outer SELECT produces. Thus, the subquery must be re-executed for every row that the outer query considers.

The query uses the COUNT function to return a value to the main query. The ORDER BY clause then orders the data. The query locates and returns the 16 rows that have the 10 latest shipping dates, as the result shows.

Figure 331. Query result

```
po_num     ship_date

4745       06/21/1998
278701     06/29/1998
429Q       06/29/1998
8052       07/03/1998
B77897     07/03/1998
LZ230      07/06/1998
B77930     07/10/1998
PC6782     07/12/1998
DM354331   07/13/1998
S22942     07/13/1998
MA003      07/16/1998
W2286      07/16/1998
Z55709     07/16/1998
C3288      07/25/1998
KF2961     07/30/1998
W9925      07/30/1998
```

If you use a correlated subquery, such as Figure 330: Query on page 1641, on a large table, you should index the **ship_date** column to improve performance. Otherwise, this SELECT statement is inefficient, because it executes the subquery once for every row of the table. For information about indexing and performance issues, see the *HCL® Informix® Administrator's Guide* and your .

You cannot use a correlated subquery in the FROM clause, however, as the following invalid example illustrates:

```
SELECT item_num, stock_num FROM items,
   (SELECT stock_num FROM catalog
       WHERE stock_num = items.item_num) AS vtab;
```

The subquery in this example fails with error -24138:

```
ALL COLUMN REFERENCES IN A TABLE EXPRESSION MUST REFER
TO TABLES IN THE FROM CLAUSE OF THE TABLE EXPRESSION.
```

The database server issues this error because the **items.item_num** column in the subquery also appears in the Projection clause of the outer query, but the FROM clause of the inner query specifies only the **catalog** table. The term *table expression* in the error message text refers to the set of column values or expressions that are returned by a subquery in the FROM clause, where only uncorrelated subqueries are valid.

The EXISTS keyword

The keyword EXISTS is known as an *existential qualifier* because the subquery is true only if the outer SELECT, as the following query shows, finds at least one row.

Figure 332. Query

```
SELECT UNIQUE manu_name, lead_time
   FROM manufact
   WHERE EXISTS
      (SELECT * FROM stock
         WHERE description MATCHES '*shoe*'
            AND manufact.manu_code = stock.manu_code);
```

You can often construct a query with EXISTS that is equivalent to one that uses IN. The following query uses an IN predicate to construct a query that returns the same result as the query above.

Figure 333. Query

```
SELECT UNIQUE manu_name, lead_time
   FROM stock, manufact
   WHERE manufact.manu_code IN
      (SELECT manu_code FROM stock
         WHERE description MATCHES '*shoe*')
            AND stock.manu_code = manufact.manu_code;
```

Figure 332: Query on page 1642 and Figure 333: Query on page 1643 return rows for the manufacturers that produce a kind of shoe, as well as the lead time for ordering the product. The result shows the return values.

Figure 334. Query result

```
manu_name       lead_time

Anza               5
Hero               4
Karsten           21
Nikolus            8
ProCycle           9
Shimara           30
```

Add the keyword NOT to IN or to EXISTS to create a search condition that is the opposite of the condition in the preceding queries. You can also substitute !=ALL for NOT IN.

The following query shows two ways to do the same thing. One way might allow the database server to do less work than the other, depending on the design of the database and the size of the tables. To find out which query might be better, use the SET EXPLAIN command to get a listing of the query plan. SET EXPLAIN is discussed in your and *HCL® Informix® Guide to SQL: Syntax*.

Figure 335. Query

```
SELECT customer_num, company FROM customer
  WHERE customer_num NOT IN
    (SELECT customer_num FROM orders
       WHERE customer.customer_num = orders.customer_num);

SELECT customer_num, company FROM customer
  WHERE NOT EXISTS
    (SELECT * FROM orders
       WHERE customer.customer_num = orders.customer_num);
```

Each statement in the query above returns the following rows, which identify customers who have not placed orders.

Figure 336. Query result

```
customer_num company

         102 Sports Spot
         103 Phil's Sports
         105 Los Altos Sports
         107 Athletic Supplies
         108 Quinn's Sports
         109 Sport Stuff
         113 Sportstown
         114 Sporting Place
         118 Blue Ribbon Sports
         125 Total Fitness Sports
         128 Phoenix University
```

The keywords EXISTS and IN are used for the set operation known as *intersection*, and the keywords NOT EXISTS and NOT IN are used for the set operation known as *difference*. These concepts are discussed in Set operations on page 1650.

The following query performs a subquery on the **items** table to identify all the items in the **stock** table that have not yet been ordered.

Figure 337. Query

```
SELECT * FROM stock
   WHERE NOT EXISTS
      (SELECT * FROM items
         WHERE stock.stock_num = items.stock_num
            AND stock.manu_code = items.manu_code);
```

The query returns the following rows.

Figure 338. Query result

```
stock_num manu_code description unit_price unit unit_descr

      101  PRC        bicycle tires       $88.00  box   4/box
      102  SHM        bicycle brakes     $220.00  case  4 sets/case
      102  PRC        bicycle brakes     $480.00  case  4 sets/case
      105  PRC        bicycle wheels      $53.00  pair  pair

      312  HRO        racer goggles       $72.00  box   12/box
      313  SHM        swim cap            $72.00  box   12/box
      313  ANZ        swim cap            $60.00  box   12/box
```

No logical limit exists to the number of subqueries a SELECT statement can have.

Perhaps you want to check whether information has been entered correctly in the database. One way to find errors in a database is to write a query that returns output only when errors exist. A subquery of this type serves as a kind of *audit query*, as the following query shows.

Figure 339. Query

```
SELECT * FROM items
   WHERE total_price != quantity *
      (SELECT unit_price FROM stock
         WHERE stock.stock_num = items.stock_num
            AND stock.manu_code = items.manu_code);
```

The query returns only those rows for which the total price of an item on an order is not equal to the stock unit price times the order quantity. If no discount has been applied, such rows were probably entered incorrectly in the database. The query returns rows only when errors occur. If information is correctly inserted into the database, no rows are returned.

Figure 340. Query result

```
item_num order_num stock_num manu_code quantity total_price

       1      1004         1 HRO              1     $960.00
       2      1006         5 NRG              5     $190.00
```

Subqueries in DELETE and UPDATE statements

Besides subqueries within the WHERE clause of a SELECT statement, you can use subqueries within other data manipulation language (DML) statements, including the WHERE clause of DELETE and UPDATE statements.

Certain restrictions apply. If the FROM clause of a subquery returns more than one row, and the clause specifies the same table or view that the outer DML statement is modifying, the DML operation will succeed under these circumstances:

- The DML statement is not an INSERT statement.
- No SPL routine within the subquery references the table that is being modified.
- The subquery does not include a correlated column name.
- The subquery is specified using the Condition with Subquery syntax in the WHERE clause of the DELETE or UPDATE statement.

If any of these conditions are not met, the DML operation fails with error -360.

The following example updates the **stock** table by increasing the **unit_price** value by 10% for a subset of prices. The WHERE clause specifies which prices to increase by applying the IN operator to the rows returned by a subquery that selects only the rows of the **stock** table where the **unit_price** value is less than 75.

```
UPDATE stock SET unit_price = unit_price * 1.1
   WHERE unit_price IN
      (SELECT unit_price FROM stock WHERE unit_price < 75);
```

Handle collections in SELECT statements

The database server provides the following SQL features to handle collection expressions:

Collection subquery

A collection subquery takes a virtual table (the result of a subquery) and converts it into a collection.

A collection subquery always returns a collection of type MULTISET. You can use a collection subquery to convert a Query result of relational data into a MULTISET collection. For information about the collection data types, see the *IBM® Informix® Database Design and Implementation Guide*.

Collection-derived table

A collection-derived table takes a collection and converts it into a virtual table.

Each element of the collection is constructed as a row in the collection-derived table. You can use a collection-derived table to access the individual elements of a collection.

The collection subquery and collection-derived table features represent inverse operations: the collection subquery converts row values from a relational table into a collection whereas the collection-derived table converts the elements of a collection into rows of a relational table.

Collection subqueries

A collection subquery enables users to construct a collection expression from a subquery expression. A collection subquery uses the MULTISET keyword immediately before the subquery to convert the values returned into a MULTISET collection. When you use the MULTISET keyword before a subquery expression, however, the database server does not change the rows of the underlying table but only modifies a copy of those rows. For example, if a collection subquery is passed to a user-defined routine that modifies the collection, then a copy of the collection is modified but not the underlying table.

A collection subquery is an expression that can take either of the following forms:

```
MULTISET(SELECT expression1, expression2... FROM tab_name...)
MULTISET(SELECT ITEM expression FROM tab_name...)
```

Omit the ITEM keyword in a collection subquery

If you omit the ITEM keyword in the collection subquery expression, the collection subquery is a MULTISET whose element type is always an unnamed ROW type. The fields of the unnamed ROW type match the data types of the expressions specified in the Projection clause of the subquery.

Suppose you create the following table that contains a column of type MULTISET:

```
CREATE TABLE tab2
(
   id_num INT,
   ms_col MULTISET(ROW(a INT) NOT NULL)
);
```

The following query shows how you might use a collection subquery in a WHERE clause to convert the rows of INT values that the subquery returns to a collection of type MULTISET. In this example, the database server returns rows when the **ms_col** column of **tab2** is equal to the result of the collection subquery expression

Figure 341. Query

```
SELECT id_num FROM tab2
   WHERE ms_col = (MULTISET(SELECT int_col FROM tab1));
```

The query omits the ITEM keyword in the collection subquery, so the INT values the subquery returns are of type MULTISET (ROW(a INT) NOT NULL) that matches the data type of the **ms_col** column of **tab2**.

Specify the ITEM keyword in a collection subquery

When the projection list of the subquery contains a single expression, you can preface the projection list of the subquery with the ITEM keyword to specify that the element type of the MULTISET matches the data type of the subquery result. In other words, when you include the ITEM keyword, the database server does not put a row wrapper around the projection list. For example, if the subquery (that immediately follows the MULTISET keyword) returns INT values, the collection subquery is of type MULTISET(INT NOT NULL).

Suppose you create a function int_func() that accepts an argument of type MULTISET(INT NOT NULL). The following query shows a collection subquery that converts rows of INT values to a MULTISET and uses the collection subquery as an argument in the function int_func().

Figure 342. Query

```
EXECUTE FUNCTION int_func(MULTISET(SELECT ITEM int_col
   FROM tab1
   WHERE int_col BETWEEN 1 AND 10));
```

The query includes the ITEM keyword in the subquery, so the **int_col** values that the query returns are converted to a collection of type MULTISET (INT NOT NULL). Without the ITEM keyword, the collection subquery would return a collection of type MULTISET (ROW(a INT) NOT NULL).

Collection subqueries in the FROM clause

Collection subqueries are valid in the FROM clause of SELECT statements, where the outer query can use the values returned by the subquery as a source of data.

The query examples in the section Collection subqueries on page 1646 specify collection subqueries by using the TABLE keyword followed (within parentheses) by the MULTISET keyword, followed by a subquery. This syntax is the HCL® Informix® extension to the ANSI/ISO standard for the SQL language.

In the FROM clause of the SELECT statement, and only in that context, you can substitute syntax that complies with the ANSI/ISO standard for SQL by specifying a subquery, omitting the TABLE and MULTISET keywords and the nested parentheses, to specify a collection subquery.

The following query uses the HCL® Informix® extension syntax to join two collection subqueries in the FROM clause of the outer query:

Figure 343. Query

```
SELECT * FROM TABLE(MULTISET(SELECT SUM(C1) FROM T1 GROUP BY C1)),
         TABLE(MULTISET(SELECT SUM(C1) FROM T2 GROUP BY C2));
```

The following logically equivalent query returns the same results as the query above by using ANSI/ISO-compliant syntax to join two derived tables in the FROM clause of the outer query:

Figure 344. Query

```
SELECT * FROM (SELECT SUM(C1) FROM T1 GROUP BY C1),
              (SELECT SUM(C1) FROM T2 GROUP BY C2);
```

An advantage of this query over the TABLE(MULTISET(SELECT ...)) HCL® Informix® extension version is that it can also be executed by any database server that supports the ANSI/ISO-compliant syntax in the FROM clause. For more information about syntax and restrictions for collection subqueries, see the *HCL® Informix® Guide to SQL: Syntax*.

Collection-derived tables

A *collection-derived table* enables you to handle the elements of a collection expression as rows in a virtual table. Use the TABLE keyword in the FROM clause of a SELECT statement to create a collection-derived table. The database server supports collection-derived tables in SELECT, INSERT, UPDATE, and DELETE statements.

The following query uses a collection-derived table named **c_table** to access elements from the **sales** column of the **sales_rep** table in the **superstores_demo** database. The **sales** column is a collection of an unnamed row type whose two fields, **month** and **amount**, store sales data. The query returns an element for **sales.amount** when **sales.month** equals 98-03. Because the inner select is itself an expression, it cannot return more than one column value per iteration of the outer query. The outer query specifies how many rows of the **sales_rep** table are evaluated.

Figure 345. Query

```
SELECT (SELECT c_table.amount FROM TABLE (sales_rep.sales) c_table
   WHERE c_table.month = '98-03')
   FROM sales_rep;
```

Figure 346. Query result

```
(expression)

$47.22
$53.22
```

The following query uses a collection-derived table to access elements from the **sales** collection column where the **rep_num** column equals 102. With a collection-derived table, you can specify aliases for the table and columns. If no table name is specified for a collection-derived table, the database server creates one automatically. This example specifies the derived column list **s_month** and **s_amount** for the collection-derived table **c_table**.

Figure 347. Query

```
SELECT * FROM TABLE((SELECT sales FROM sales_rep
   WHERE sales_rep.rep_num = 102)) c_table(s_month, s_amount);
```

Figure 348. Query result

```
s_month         s_amount

1998-03           $53.22
1998-04           $18.22
```

The following query creates a collection-derived table but does not specify a derived table or derived column names. The query returns the same result as Figure 347: Query on page 1648 except the derived columns assume the default field names of the **sales** column in the **sales_rep** table.

Figure 349. Query

```
SELECT * FROM TABLE((SELECT sales FROM sales_rep
   WHERE sales_rep.rep_num = 102));
```

Figure 350. Query result

```
month             amount

1998-03           $53.22
1998-04           $18.22
```

Restriction: A collection-derived table is read-only, so it cannot be the target table of INSERT, UPDATE, or DELETE statements or the underlying table of an updatable cursor or view.

For a complete description of the syntax and restrictions on collection-derived tables, see the *HCL® Informix® Guide to SQL: Syntax*.

ISO-compliant syntax for collection derived tables

The query examples in the topic Collection-derived tables on page 1648 specify collection-derived tables by using the TABLE keyword followed by a SELECT statement enclosed within double parentheses. This syntax is the HCL® Informix® extension to the ANSI/ISO standard for the SQL language.

In the FROM clause of the SELECT statement, however, and only in that context, you can instead use syntax that complies with the ANSI/ISO standard for SQL by specifying a subquery, without the TABLE keyword or the nested parentheses, to define a collection-derived table.

The following example is logically equivalent to Figure 347: Query on page 1648, and specifies the derived column list **s_month** and **s_amount** for the collection-derived table **c_table**.

Figure 351. Query

```
SELECT * FROM (SELECT sales FROM sales_rep
   WHERE sales_rep.rep_num = 102) c_table(s_month, s_amount);
```

Figure 352. Query result

```
s_month         s_amount

1998-03           $53.22
1998-04           $18.22
```

As in the HCL® Informix® extension syntax, declaring names for the derived table or for its columns is optional, rather than required. The following query uses ANSI/ISO-compliant syntax for a derived table in the FROM clause of the outer query, and produces the same results as Figure 349: Query on page 1649:

Figure 353. Query

```
SELECT * FROM (SELECT sales FROM sales_rep
   WHERE sales_rep.rep_num = 102);
```

Figure 354. Query result

```
month            amount

1998-03          $53.22
1998-04          $18.22
```

Set operations

The standard set operations *union*, *intersection*, and *difference* let you manipulate database information. These three operations let you use SELECT statements to check the integrity of your database after you perform an update, insert, or delete. They can be useful when you transfer data to a history table, for example, and want to verify that the correct data is in the history table before you delete the data from the original table.

Union

A union operation uses the UNION operator to combine two queries into a single *compound query*. You can use the UNION operator between two or more SELECT statements to produce a temporary table that contains rows that exist in any or all of the original tables. You can also use the UNION operator in the definition of a view.

You cannot use the UNION operator inside a subquery in the following contexts

- in the Projection clause of the SELECT statement
- in the WHERE clause of the SELECT, INSERT, DELETE, or UPDATE statement.

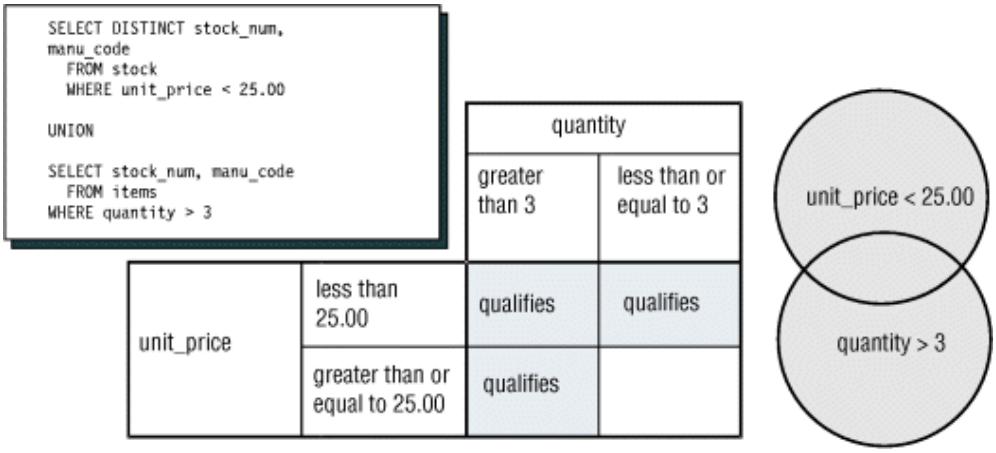
The UNION operator is valid, however, in a subquery in the FROM clause of the SELECT statement, as in the following example:

```
SELECT * FROM (SELECT col1 FROM tab1 WHERE col1 = 100) AS vtab1(c1),
   (SELECT col1 FROM tab2 WHERE col1 = 10
    UNION ALL
    SELECT col1 FROM tab1 WHERE col1 < 50 ) AS vtab2(vc1);
```

HCL Informix® does not support ordering on ROW types. Because a UNION operation requires a sort to remove duplicate values, you cannot use a UNION operator when either query in the union operation includes ROW type data. However, the database server does support UNION ALL with ROW type data, because this type of operation does not require a sort.

The following figure illustrates the UNION set operation.

Figure 355. The Union set operation



The UNION keyword selects all rows from the two queries, removes duplicates, and returns what is left. Because the results of the queries are combined into a single result, the projection list in each query must have the same number of columns. Also, the corresponding columns that are selected from each table must contain compatible data types (CHARACTER data type columns must be the same length), and these corresponding columns must all allow or all disallow NULL values.

For the complete syntax of the SELECT statement and the UNION operator, see the *HCL® Informix® Guide to SQL: Syntax*. For information specific to the product and any limitations that involve the INTO clause and compound queries, see the *HCL® Informix® Enterprise Replication Guide*.

The following query performs a union on the **stock_num** and **manu_code** columns in the **stock** and **items** tables.

Figure 356. Query

```
SELECT DISTINCT stock_num, manu_code FROM stock
   WHERE unit_price < 25.00
UNION
SELECT stock_num, manu_code FROM items
   WHERE quantity > 3;
```

The query selects those items that have a unit price of less than $25.00 or that have been ordered in quantities greater than three and lists their **stock_num** and **manu_code**, as the result shows.

Figure 357. Query result

```
stock_num manu_code

        5 ANZ
        5 NRG
        5 SMT
        9 ANZ
      103 PRC
      106 PRC
      201 NKL
      301 KAR
      302 HRO
      302 KAR
```

ORDER BY clause with UNION

As the following query shows, when you include an ORDER BY clause, it must follow the final SELECT statement and use an integer, not an identifier, to refer to the ordering column. Ordering takes place after the set operation is complete.

Figure 358. Query

```
SELECT DISTINCT stock_num, manu_code FROM stock
   WHERE unit_price < 25.00
UNION
SELECT stock_num, manu_code FROM items
   WHERE quantity > 3
   ORDER BY 2;
```

The compound query above selects the same rows as Figure 356: Query on page 1651 but displays them in order of the manufacturer code, as the result shows.

Figure 359. Query result

```
stock_num manu_code

        5 ANZ
        9 ANZ
      302 HRO
      301 KAR
      302 KAR
      201 NKL
        5 NRG
      103 PRC
      106 PRC
        5 SMT
```

The UNION ALL keywords

By default, the UNION keyword excludes duplicate rows. To retain the duplicate values, add the optional keyword ALL, as the following query shows.

Figure 360. Query

```
SELECT stock_num, manu_code FROM stock
   WHERE unit_price < 25.00
UNION ALL
SELECT stock_num, manu_code FROM items
   WHERE quantity > 3
   ORDER BY 2
   INTO TEMP stock item;
```

The query uses the UNION ALL keywords to unite two SELECT statements and adds an INTO TEMP clause after the final SELECT to put the results into a temporary table. It returns the same rows as Figure 358: Query on page 1652 but also includes duplicate values.

Figure 361. Query result

```
stock_num manu_code

        9 ANZ
        5 ANZ
        9 ANZ
        5 ANZ
        9 ANZ
       ⋮
        5 NRG
        5 NRG
      103 PRC
      106 PRC
        5 SMT
        5 SMT
```

Different column names

Corresponding columns in the Projection clauses for the combined queries must have compatible data types, but the columns do not need to use the same column names.

The following query selects the **state** column from the **customer** table and the corresponding **code** column from the **state** table.

Figure 362. Query

```
SELECT DISTINCT state FROM customer
   WHERE customer_num BETWEEN 120 AND 125
UNION
SELECT DISTINCT code FROM state
   WHERE sname MATCHES '*a';
```

The query returns state code abbreviations for customer numbers 120 through 125 and for states whose **sname** ends in `a`.

Figure 363. Query result

```
state

AK
AL
AZ
CA
DE

SD
VA
WV
```

In compound queries, the column names or display labels in the first SELECT statement are the ones that appear in the results. Thus, in the query, the column name **state** from the first SELECT statement is used instead of the column name **code** from the second.

UNION with multiple tables

The following query performs a union on three tables. The maximum number of unions depends on the practicality of the application and any memory limitations.

Figure 364. Query

```
SELECT stock_num, manu_code FROM stock
   WHERE unit_price > 600.00
UNION ALL
SELECT stock_num, manu_code FROM catalog
   WHERE catalog_num = 10025
UNION ALL
SELECT stock_num, manu_code FROM items
   WHERE quantity = 10
   ORDER BY 2;
```

The query selects items where the **unit_price** in the **stock** table is greater than $600, the **catalog_num** in the **catalog** table is 10025, or the **quantity** in the **items** table is 10; and the query orders the data by **manu_code**. The result shows the return values.

Figure 365. Query result

```
stock_num manu_code

        5 ANZ
        9 ANZ
        8 ANZ
        4 HSK
        1 HSK
      203 NKL
        5 NRG
      106 PRC
      113 SHM
```

A literal in the Projection clause

The following query uses a literal in the projection list to tag the output of part of a union so it can be distinguished later. The tag is given the label **sortkey**. The query uses **sortkey** to order the retrieved rows.

Figure 366. Query

```
SELECT '1' sortkey, lname, fname, company,
     city, state, phone
   FROM customer x
   WHERE state = 'CA'
UNION
SELECT '2' sortkey, lname, fname, company,
     city, state, phone
   FROM customer y
   WHERE state <> 'CA'
   INTO TEMP calcust;
SELECT * FROM calcust
   ORDER BY 1;
```

The query creates a list in which the customers from California appear first.

Figure 367. Query result

```
sortkey  1
lname    Baxter
fname    Dick
company  Blue Ribbon Sports
city     Oakland
state    CA
phone    415-655-0011

sortkey  1
lname    Beatty
fname    Lana
company  Sportstown
city     Menlo Park
state    CA
phone    415-356-9982
⋮
sortkey  2
lname    Wallack
fname    Jason
company  City Sports
city     Wilmington
state    DE
phone    302-366-7511
```

A FIRST clause

You can use the FIRST clause to select the first rows that result from a union query. The following query uses a FIRST clause to return the first five rows of a union between the **stock** and **items** tables.

Figure 368. Query

```
SELECT FIRST 5 DISTINCT stock_num, manu_code
   FROM stock
   WHERE unit_price < 55.00
UNION
SELECT stock_num, manu_code
   FROM items
   WHERE quantity > 3;
```

Figure 369. Query result

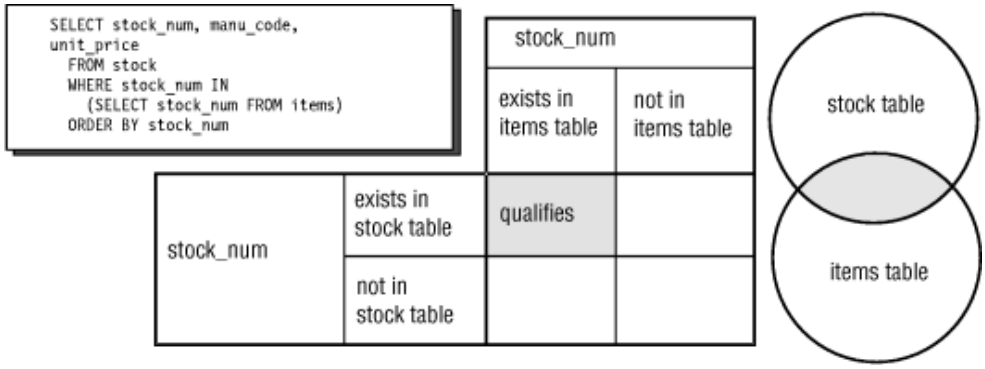
```
stock_num manu_code

        5 NRG
        5 ANZ
        6 SMT
        6 ANZ
        9 ANZ
```

Intersection

The *intersection* of two sets of rows produces a table that contains rows that exist in both the original tables. Use the keyword EXISTS or IN to introduce subqueries that show the intersection of two sets. The following figure illustrates the intersection set operation.

Figure 370. The intersection set operation



The following query is an example of a nested SELECT statement that shows the intersection of the **stock** and **items** tables. The result contains all the elements that appear in both sets and returns the following rows.

Figure 371. Query

```
SELECT stock_num, manu_code, unit_price FROM stock
   WHERE stock_num IN
      (SELECT stock_num FROM items)
   ORDER BY stock_num;
```

Figure 372. Query result

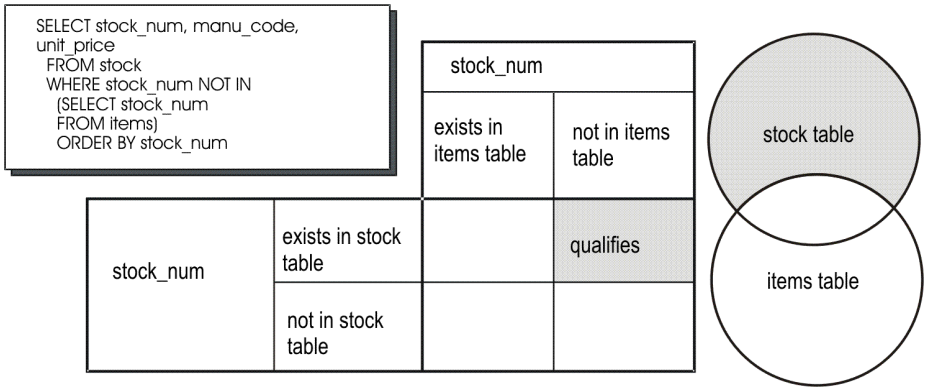
```
stock_num manu_code unit_price

        1 HRO          $250.00
        1 HSK          $800.00
        1 SMT          $450.00
        2 HRO          $126.00
        3 HSK          $240.00
        3 SHM          $280.00
       .
      306 SHM          $190.00
      307 PRC          $250.00
      309 HRO           $40.00
      309 SHM           $40.00
```

Difference

The *difference* between two sets of rows produces a table that contains rows in the first set that are not also in the second set. Use the keywords NOT EXISTS or NOT IN to introduce subqueries that show the difference between two sets. The following figure illustrates the difference set operation.

Figure 373. The difference set operation



The following query is an example of a nested SELECT statement that shows the difference between the **stock** and **items** tables.

Figure 374. Query

```
SELECT stock_num, manu_code, unit_price FROM stock
   WHERE stock_num NOT IN
      (SELECT stock_num FROM items)
   ORDER BY stock_num;
```

The result contains all the elements from only the first set, which returns 17 rows.

Figure 375. Query result

```
stock_num manu_code unit_price

      102 PRC          $480.00
      102 SHM          $220.00
      106 PRC           $23.00
      ⋮
      312 HRO           $72.00
      312 SHM           $96.00
      313 ANZ           $60.00
      313 SHM           $72.00
```

Summary

This chapter builds on concepts introduced in Compose SELECT statements on page 1532. It provides sample syntax and results for more advanced kinds of SELECT statements, which are used to query a relational database. This chapter presents the following material:

- Introduces the GROUP BY and HAVING clauses, which you can use with aggregates to return groups of rows and apply conditions to those groups
- Shows how to join a table to itself with a self-join to compare values in a column with other values in the same column and to identify duplicates
- Explains how an outer join treats two or more tables asymmetrically, and provides examples of the four kinds of outer join using both the HCL® Informix® extension and ANSI join syntax.

- Describes how to nest a SELECT statement in the WHERE clause of another SELECT statement to create correlated and uncorrelated subqueries and shows how to use aggregate functions in subqueries
- Describes how to nest SELECT statements in the FROM clause of another SELECT statement to specify uncorrelated subqueries whose results are a data source for the outer SELECT statement
- Demonstrates how to use the keywords ALL, ANY, EXISTS, IN, and SOME to create subqueries, and the effect of adding the keyword NOT or a relational operator
- Describes how to use collection subqueries to convert relational data to a collection of type MULTISET and how to use collection-derived tables to access elements within a collection
- Discusses the union, intersection, and difference set operations
- Shows how to use the UNION and UNION ALL keywords to create compound queries that consist of two or more SELECT statements

Modify data

This section describes how to modify the data in your databases. Modifying data is fundamentally different from querying data. Querying data involves examining the contents of tables. To modify data involves changing the contents of tables.

Modify data in your database

The following statements modify data:

- DELETE
- INSERT
- MERGE
- UPDATE

Although these SQL statements are relatively simple when compared with the more advanced SELECT statements, use them carefully because they change the contents of the database.

Think about what happens if the system hardware or software fails during a query. Even if the effect on the application is severe, the database itself is unharmed. However, if the system fails while a modification is under way, the state of the database is in doubt. Obviously, a database in an uncertain state has far-reaching implications. Before you delete, insert, or update rows in a database, ask yourself the following questions:

- Is user access to the database and its tables secure; that is, are specific users given limited database and table-level privileges?
- Does the modified data preserve the existing integrity of the database?
- Are systems in place that make the database relatively immune to external events that might cause system or hardware failures?

If you cannot answer yes to each of these questions, do not panic. Solutions to all these problems are built into the HCL® Informix® database servers. After a description of the statements that modify data, this section discusses these solutions. The *IBM® Informix® Database Design and Implementation Guide* covers these topics in greater detail.

Delete rows

The DELETE statement removes any row or combination of rows from a table. You cannot recover a deleted row after the transaction is committed. (Transactions are discussed under Interrupted modifications on page 1691. For now, think of a transaction and a statement as the same thing.)

When you delete a row, you must also be careful to delete any rows of other tables whose values depend on the deleted row. If your database enforces referential constraints, you can use the ON DELETE CASCADE option of the CREATE TABLE or ALTER TABLE statements to allow deletes to cascade from one table in a relationship to another. For more information on referential constraints and the ON DELETE CASCADE option, refer to Referential integrity on page 1682.

Delete all rows of a table

The DELETE statement specifies a table and usually contains a WHERE clause that designates the row or rows that are to be removed from the table. If the WHERE clause is left out, all rows are deleted.

Important: Do not execute the following statement.

```
DELETE FROM customer;
```

You can write DELETE statements with or without the FROM keyword.

```
DELETE customer;
```

Because these DELETE statements do not contain a WHERE clause, all rows from the **customer** table are deleted. If you attempt an unconditional delete using the menu options, the program warns you and asks for confirmation. However, an unconditional DELETE from within a program can occur without warning.

If you want to delete rows from a table named **from**, you must first set the **DELIMIDENT** environment variable, or qualify the name of the table with the name of its owner:

```
DELETE legree.from;
```

For more information about delimited identifiers and **DELIMIDENT** environment variable, see the descriptions of the Quoted String expression and of the Identifier segment in the *HCL® Informix® Guide to SQL: Syntax*.

Delete all rows using TRUNCATE

You can use the TRUNCATE statement to quickly remove all rows from a table and also remove all corresponding index data. You cannot recover deleted rows after the transaction is committed. You can use the TRUNCATE statement on tables that contain any type of columns, including smart large objects.

Removing rows with the TRUNCATE statement is faster than removing them with the DELETE statement. It is not necessary to run the UPDATE STATISTICS statement immediately after the TRUNCATE statement. After TRUNCATE executes successfully, HCL Informix® automatically updates the statistics and distributions for the table and for its indexes in the system catalog to show no rows in the table or in its dbspace partitions.

For a description of logging, see Transaction logging on page 1692.

TRUNCATE is a data-definition language statement that does not activate DELETE triggers, if any are defined on the table. For an explanation on using triggers, see Create and use triggers on page 1821.

If the table that the TRUNCATE statement specifies is a typed table, a successful TRUNCATE operation removes all the rows and B-tree structures from that table and from all its subtables within the table hierarchy. TRUNCATE has no equivalent to the ONLY keyword of the DELETE statement to restricts the operation to a single table within the typed table hierarchy.

always logs the TRUNCATE operation, even for a non-logging table. In databases that support transaction logging, only the COMMIT WORK or ROLLBACK WORK statement of SQL is valid after TRUNCATE within the same transaction. For information on the performance impact of using the TRUNCATE statement, see your . For the complete syntax, see the *HCL® Informix® Guide to SQL: Syntax*.

Delete specified rows

The WHERE clause in a DELETE statement has the same form as the WHERE clause in a SELECT statement. You can use it to designate exactly which row or rows should be deleted. You can delete a customer with a specific customer number, as the following example shows:

```
DELETE FROM customer WHERE customer_num = 175;
```

In this example, because the **customer_num** column has a unique constraint, you can ensure that no more than one row is deleted.

Delete selected rows

You can also choose rows that are based on nonindexed columns, as the following example shows:

```
DELETE FROM customer WHERE company = 'Druid Cyclery';
```

Because the column that is tested does not have a unique constraint, this statement might delete more than one row. (Druid Cyclery might have two stores, both with the same name but different customer numbers.)

To find out how many rows a DELETE statement affects, select the count of qualifying rows from the **customer** table for Druid Cyclery.

```
SELECT COUNT(*) FROM customer WHERE company = 'Druid Cyclery';
```

You can also select the rows and display them to ensure that they are the ones you want to delete.

Using a SELECT statement as a test is only an approximation, however, when the database is available to multiple users concurrently. Between the time you execute the SELECT statement and the subsequent DELETE statement, other users could have modified the table and changed the result. In this example, another user might perform the following actions:

- Insert a new row for another customer named Druid Cyclery
- Delete one or more of the Druid Cyclery rows before you insert the new row
- Update a Druid Cyclery row to have a new company name, or update some other customer to have the name Druid Cyclery.

Although it is not likely that other users would do these things in that brief interval, the possibility does exist. This same problem affects the UPDATE statement. Ways of addressing this problem are discussed under Concurrency and locks on page 1694, and in greater detail in Programming for a multiuser environment on page 1733.

Another problem you might encounter is a hardware or software failure before the statement finishes. In this case, the database might have deleted no rows, some rows, or all specified rows. The state of the database is unknown, which is undesirable. To prevent this situation, use transaction logging, as Interrupted modifications on page 1691 discusses.

Delete rows that contain row types

When a row contains a column that is defined on a ROW type, you can use dot notation to specify that the only rows deleted are those that contain a specific field value. For example, the following statement deletes only those rows from the **employee** table in which the value of the **city** field in the **address** column is `San Jose`:

```
DELETE FROM employee
   WHERE address.city = 'San Jose';
```

In the preceding statement, the **address** column might be a named ROW type or an unnamed ROW type. The syntax you use to specify field values of a ROW type is the same.

Delete rows that contain collection types

When a row contains a column that is defined on a collection type, you can search for a particular element in a collection and delete the row or rows in which that element is found. For example, the following statement deletes rows in which the **direct_reports** column contains a collection with the element `Baker`:

```
DELETE FROM manager
   WHERE 'Baker' IN direct_reports;
```

Delete rows from a supertable

When you delete the rows of a supertable, the scope of the delete is a supertable and its subtables. Suppose you create a supertable **person** that has two subtables **employee** and **sales_rep** defined under it. The following DELETE statement on the **person** table can delete rows from all the tables **person**, **employee**, and **sales_rep**:

```
DELETE FROM person
   WHERE name ='Walker';
```

To limit a delete to rows of the supertable only, you must use the ONLY keyword in the DELETE statement. For example, the following statement deletes rows of the **person** table only:

```
DELETE FROM ONLY(person)
   WHERE name ='Walker';
```

Important: Use caution when you delete rows from a supertable because the scope of a delete on a supertable includes the supertable and all its subtables.

Complicated delete conditions

The WHERE clause in a DELETE statement can be almost as complicated as the one in a SELECT statement. It can contain multiple conditions that are connected by AND and OR, and it might contain subqueries.

Suppose you discover that some rows of the **stock** table contain incorrect manufacturer codes. Rather than update them, you want to delete them so that they can be re-entered. You know that these rows, unlike the correct ones, have no matching rows in the **manufact** table. The fact that these incorrect rows have no matching rows in the **manufact** table allows you to write a DELETE statement such as the one in the following example:

```
DELETE FROM stock
   WHERE 0 = (SELECT COUNT(*) FROM manufact
        WHERE manufact.manu_code = stock.manu_code);
```

The subquery counts the number of rows of **manufact** that match; the count is 1 for a correct row of **stock** and 0 for an incorrect one. The latter rows are chosen for deletion.

Tip: One way to develop a DELETE statement with a complicated condition is to first develop a SELECT statement that returns precisely the rows to be deleted. Write it as `SELECT *`; when it returns the desired set of rows, change `SELECT *` to read `DELETE` and execute it once more.

The WHERE clause of a DELETE statement cannot use a subquery that tests the same table. That is, when you delete from **stock**, you cannot use a subquery in the WHERE clause that also selects from **stock**.

The key to this rule is in the FROM clause. If a table is named in the FROM clause of a DELETE statement, it cannot also appear in the FROM clause of a subquery of the DELETE statement.

The Delete clause of MERGE

Instead of writing a subquery in the WHERE clause, you can use the MERGE statement to join rows from a source tables and a target table, and then delete from the target the rows that match the join condition. (The source table in a Delete MERGE can also be a collection-derived table whose rows are the result of a query that joins other tables and views, but in the example that follows, the source is a single table.)

As in the previous example, suppose you discover that some rows of the **stock** table contain incorrect manufacturer codes. Rather than update them, you want to delete them so that they can be re-entered. You can use the MERGE statement that specifies **stock** as the target table, **manufact** as the source table, a join condition in the ON clause, and with the Delete clause for the **stock** rows with incorrect manufacturer codes, as in the following example:

```
MERGE INTO stock USING manufact
   ON stock.manu_code != manufact.manu_code
WHEN MATCHED THEN DELETE;
```

In this example, all the rows of the **stock** table for which the join condition in the ON clause is satisfied will be deleted. Here the inequality predicate in the join condition (`stock.manu_code != manufact.manu_code`) evaluates to true for the rows of **stock** in which the **manu_code** column value is not equal to any **manu_code** value in the **manufact** table.

The **source** table that is being joined to the *target* table must be listed in the USING clause.

The MERGE statement can also update rows of the target table, or insert data from the source table into the target table, according to whether or not the row satisfies the condition that the ON clause specifies for joining the target and source tables. A single MERGE statement can also combine both DELETE and INSERT operations, or can combine both UPDATE and INSERT operations without deleting any rows. The source table is unchanged by the MERGE statement. For more information on the syntax and restrictions for Delete merges, Insert merges, and Update merges, see the description of the MERGE statement in the *HCL® Informix® Guide to SQL: Syntax*.

Insert rows

The INSERT statement adds a new row, or rows, to a table. The statement has two basic functions. It can create a single new row using column values you supply, or it can create a group of new rows using data selected from other tables.

Single rows

In its simplest form, the INSERT statement creates one new row from a list of column values and puts that row in the table. The following statement shows how to add a row to the **stock** table:

```
INSERT INTO stock
   VALUES (115, 'PRC', 'tire pump', 108, 'box', '6/box');
```

The **stock** table has the following columns:

stock_num

A number that identifies the type of merchandise.

manu_code

A foreign key to the **manufact** table.

description

A description of the merchandise.

unit_price

The unit price of the merchandise.

unit

The unit of measure

unit_descr

Characterizes the unit of measure.

The values that are listed in the VALUES clause in the preceding example have a one-to-one correspondence with the columns of the **stock** table. To write a VALUES clause, you must know the columns of the tables as well as their sequence from first to last.

Possible column values

The VALUES clause accepts only constant values, not general SQL expressions. You can supply the following values:

- Literal numbers
- Literal DATETIME values
- Literal INTERVAL values
- Quoted strings of characters
- The word NULL for a NULL value
- The word TODAY for the current date
- The word CURRENT (or SYSDATE) for the current date and time
- The word USER for your authorization identifier
- The word DBSERVERNAME (or SITENAME) for the name of the computer where the database server is running

Note: An alternative to the INSERT statement is the MERGE statement, which can use the same VALUES clause syntax as the INSERT statement to insert rows into a table. The MERGE statement performs an outer join of a source table and a target table, and then inserts into the target table any rows in the result set of the join for which the join predicate evaluates to FALSE. The source table is unchanged by the MERGE statement. Besides inserting rows, the MERGE statement can optionally combine both DELETE and INSERT operations, or combine both UPDATE and INSERT operations. For more information about the syntax and the restrictions on Insert merges, Delete merges, and Update merges, see the description of the MERGE statement in the *HCL® Informix® Guide to SQL: Syntax*.

Restrictions on column values

Some columns of a table might not allow null values. If you attempt to insert NULL in such a column, the statement is rejected. Other columns in the table might not permit duplicate values. If you specify a value that is a duplicate of one that is already in such a column, the statement is rejected. Some columns might even restrict the possible column values allowed. Use data integrity constraints to restrict columns. For more information, see Data integrity on page 1680.

Restriction: Do not specify the currency symbols for columns that contain money values. Just specify the numeric value of the amount.

The database server can convert between numeric and character data types. You can give a string of numeric characters (for example, `'-0075.6'`) as the value of a numeric column. The database server converts the numeric string to a number. An error occurs only if the string does not represent a number.

You can specify a number or a date as the value for a character column. The database server converts that value to a character string. For example, if you specify TODAY as the value for a character column, a character string that represents the current date is used. (The **DBDATE** environment variable specifies the format that is used.)

Serial data types

A table can have only one column of the SERIAL data type. It can also have either a SERIAL8 column or a BIGSERIAL column.

When you insert values, specify the value zero for the serial column. The database server generates the next actual value in sequence. Serial columns do not allow NULL values.

You can specify a nonzero value for a serial column (as long as it does not duplicate any existing value in that column), and the database server uses the value. That nonzero value might set a new starting point for values that the database server generates. (The next value the database server generates for you is one greater than the maximum value in the column.)

List specific column names

You do not have to specify values for every column. Instead, you can list the column names after the table name and then supply values for only those columns that you named. The following example shows a statement that inserts a new row into the **stock** table:

```
INSERT INTO stock (stock_num,  description,  unit_price,  manu_code)
   VALUES (115,  'tyre pump ',  114,  'SHM');
```

Only the data for the stock number, description, unit price, and manufacturer code is provided. The database server supplies the following values for the remaining columns:

- It generates a serial number for an unlisted serial column.
- It generates a default value for a column with a specific default associated with it.
- It generates a NULL value for any column that allows nulls but it does not specify a default value for any column that specifies NULL as the default value.

  You must list and supply values for all columns that do not specify a default value or do not permit NULL values.

You can list the columns in any order, as long as the values for those columns are listed in the same order. For information about how to designate null or default values for a column, see the *IBM® Informix® Database Design and Implementation Guide*.

After the INSERT statement in the preceding example is executed, the following new row is inserted into the **stock** table:

```
stock_num manu_code  description  unit_price unit unit_descr

      115       SHM  tyre pump           114
```

Both **unit** and **unit_descr** are blank, which indicates that NULL values exist in those two columns. Because the **unit** column permits NULL values, the number of tire pumps that can be purchased for $114 is not known. Of course, if a default value of `box` were specified for this column, then `box` would be the unit of measure. In any case, when you insert values into specific columns of a table, pay attention to what data is needed for that row.

Insert rows into typed tables

You can insert rows into a typed table in the same way you insert rows into a table not based on a ROW type.

When a typed table contains a row-type column (the named ROW type that defines the typed table contains a nested ROW type), you insert into the row-type column in the same way you insert into a row-type column for a table not based on a ROW type. The following section, Syntax rules for inserts on columns on page 1666, describes how to perform inserts into row-type columns.

This section uses row types **zip_t**, **address_t**, and **employee_t** and typed table **employee** for examples. The following figure shows the SQL syntax that creates the row types and table.

Figure 376. SQL syntax that creates the row types and table.

```
CREATE ROW TYPE zip_t
(
   z_code    CHAR(5),
   z_suffix  CHAR(4)
);

CREATE ROW TYPE address_t
(
   street    VARCHAR(20),
   city      VARCHAR(20),
   state     CHAR(2),
   zip       zip_t
);

CREATE ROW TYPE employee_t
(
   name      VARCHAR(30),
   address   address_t,
   salary    INTEGER
);

CREATE TABLE employee OF TYPE employee_t;
```

Syntax rules for inserts on columns

The following syntax rules apply for inserts on columns that are defined on named ROW types or unnamed ROW types:

- Specify the ROW constructor before the field values to be inserted.
- Enclose the field values of the ROW type in parentheses.
- Cast the ROW expression to the appropriate named ROW type (for named ROW types).

Rows that contain named row types

The following statement shows you how to insert a row into the **employee** table in Figure 377: Create the student table. on page 1667:

```
INSERT INTO employee
   VALUES ('Poole, John',
   ROW('402 High St', 'Willits', 'CA',
   ROW(69055,1450))::address_t, 35000 );
```

Because the **address** column of the **employee** table is a named ROW type, you must use a cast operator and the name of the ROW type (**address_t**) to insert a value of type **address_t**.

Rows that contain unnamed row types

Suppose you create the table that the following figure shows. The **student** table defines the **s_address** column as an unnamed row type.

Figure 377. Create the student table.

```
CREATE TABLE student
(
s_name     VARCHAR(30),
s_address  ROW(street VARCHAR (20), city VARCHAR(20),
               state CHAR(2), zip VARCHAR(9)),
               grade_point_avg DECIMAL(3,2)
);
```

The following statement shows you how to add a row to the **student** table. To insert into the unnamed row-type column **s_address**, use the ROW constructor but do not cast the row-type value.

```
INSERT INTO student
   VALUES ('Keene, Terry',
      ROW('53 Terra Villa', 'Wheeling', 'IL', '45052'),
      3.75);
```

Specify NULL values for row types

The fields of a row-type column can contain NULL values. You can specify NULL values either at the level of the column or the field.

The following statement specifies a NULL value at the column level to insert NULL values for all fields of the **s_address** column. When you insert a NULL value at the column level, do not include the ROW constructor.

```
INSERT INTO student VALUES ('Brauer, Howie', NULL, 3.75);
```

When you insert a NULL value for particular fields of a ROW type, you must include the ROW constructor. The following INSERT statement shows how you might insert NULL values into particular fields of the **address** column of the **employee** table. (The **address** column is defined as a named ROW type.)

```
INSERT INTO employee
   VALUES (
      'Singer, John',
      ROW(NULL, 'Davis', 'CA',
      ROW(97000, 2000))::address_t, 67000
      );
```

When you specify a NULL value for the field of a ROW type, you do not need to explicitly cast the NULL value when the ROW type occurs in an INSERT statement, an UPDATE statement, or a program variable assignment.

The following INSERT statement shows how you insert NULL values for the **street** and **zip** fields of the **s_address** column for the **student** table:

```
INSERT INTO student
   VALUES(
      'Henry, John',
```

```
      ROW(NULL, 'Seattle', 'WA', NULL), 3.82
      );
```

Insert rows into supertables

No special considerations exist when you insert a row into a supertable. An INSERT statement applies only to the table that is specified in the statement. For example, the following statement inserts values into the supertable but does not insert values into any subtables:

```
INSERT INTO person
   VALUES (
      'Poole, John',
      ROW('402 Saphire St.', 'Elmondo', 'CA', '69055'),
      345605900
      );
```

Insert collection values into columns

This section describes how to insert a collection value into a column with . It does not discuss how to insert individual elements into a collection column. To access or modify the individual elements of a collection, use an SPL routine or program. For information about how to create programs to insert into a collection, see the *HCL® Informix® Enterprise Replication Guide*. For information about how to create an SPL routine to insert into a collection, see Create and use SPL routines on page 1753.

The examples that this section provides are based on the **manager** table in the following figure. The **manager** table contains both simple and nested collection types.

Figure 378. Create the manager table.

```
CREATE TABLE manager
(
  mgr_name        VARCHAR(30),
  department      VARCHAR(12),
  direct_reports  SET(VARCHAR(30) NOT NULL),
  projects        LIST(ROW(pro_name VARCHAR(15),
                    pro_members SET(VARCHAR(20) NOT NULL))
                    NOT NULL)
);
```

Insert values into simple collections and nested collections

When you insert values into a row that contains a collection column, you insert the values of all the elements that the collection contains as well as values for the other columns. For example, the following statement inserts a single row into the **manager** table, which includes columns for both simple collections and nested collections:

```
INSERT INTO manager(mgr_name, department,
   direct_reports, projects)
   VALUES
(
'Sayles', 'marketing',
"SET{'Simonian', 'Waters', 'Adams', 'Davis', 'Jones'}",
LIST{
   ROW('voyager_project', SET{'Simonian', 'Waters',
```

```
   'Adams', 'Davis'}),
   ROW ('horizon_project', SET{'Freeman', 'Jacobs',
   'Walker', 'Smith', 'Cannan'}),
   ROW ('saphire_project', SET{'Villers', 'Reeves',
   'Doyle', 'Strongin'})
   }
);
```

Insert NULL values into a collection that contains a row type

To insert values into a collection that is a ROW type, you must specify a value for each field in the ROW type.

In general, NULL values are not allowed in a collection. However, if the element type of the collection is a ROW type, you can insert NULL values into individual fields of the row type.

You can also specify an *empty collection*. An empty collection is a collection that contains no elements. To specify an empty collection, use the braces ({}). For example, the following statement inserts data into a row in the **manager** table but specifies that the **direct_reports** and **projects** columns are empty collections:

```
INSERT INTO manager
   VALUES ('Sayles', 'marketing', "SET{}",
   "LIST{ROW(NULL, SET{})}"
);
```

A collection column cannot contain NULL elements. The following statement returns an error because NULL values are specified as elements of collections:

```
INSERT INTO manager
   VALUES ('Cole', 'accounting', "SET{NULL}",
   "LIST{ROW(NULL, ""SET{NULL}"")}"
```

The following syntax rules apply for performing inserts and updates on collection types:

- Use braces ({}) to demarcate the elements that each collection contains.
- If the collection is a nested collection, use braces ({}) to demarcate the elements of both the inner and outer collections.

Insert smart large objects

When you use the INSERT statement to insert an object into a **BLOB** or **CLOB** column, the database server stores the object in an sbspace, rather than the table. The database server provides SQL functions that you can call from within an INSERT statement to import and export BLOB or CLOB data, otherwise known as smart large objects. For a description of these functions, see Smart large object functions on page 1603.

The following INSERT statement uses the filetoblob() and filetoclob() functions to insert a row of the **inmate** table. (Figure 231: Create the inmate and fbi_list tables. on page 1604 defines the **inmate** table.)

```
INSERT INTO inmate
   VALUES (437, FILETOBLOB('datafile', 'client'),
      FILETOCLOB('tmp/text', 'server'));
```

In the preceding example, the first argument for the FILETOBLOB() and FILETOCLOB() functions specifies the path of the source file to be copied into the **BLOB** and **CLOB** columns of the **inmate** table, respectively. The second argument for each function specifies whether the source file is located on the client computer ('client') or server computer ('server'). To specify the path of a file name in the function argument, apply the following rules:

- If the source file resides on the server computer, you must specify the full path name to the file (not the path name relative to the current working directory).
- If the source file resides on the client computer, you can specify either the full or relative path name to the file.



Note: For CLOB columns, direct strings can also be used in place of FILETOCLOB() function.

Multiple rows and expressions

The other major form of the INSERT statement replaces the VALUES clause with a SELECT statement. This feature allows you to insert the following data:

- Multiple rows with only one statement (each time the SELECT statement returns a row, a row is inserted)
- Calculated values (the VALUES clause permits only constants) because the projection list can contain expressions

For example, suppose a follow-up call is required for every order that has been paid for but not shipped. The INSERT statement in the following example finds those orders and inserts a row in **cust_calls** for each order:

```
INSERT INTO cust_calls (customer_num, call_descr)
   SELECT customer_num, order_num FROM orders
      WHERE paid_date IS NOT NULL
      AND ship_date IS NULL;
```

This SELECT statement returns two columns. The data from these columns (in each selected row) is inserted into the named columns of the **cust_calls** table. Then an order number (from **order_num**, a SERIAL column) is inserted into the call description, which is a character column. Remember that the database server allows you to insert integer values into a character column. It automatically converts the serial number to a character string of decimal digits.

Restrictions on the insert selection

The following list contains the restrictions on the SELECT statement for inserting rows:

- It cannot contain an INTO clause.
- It cannot contain an INTO TEMP clause.
- It cannot contain an ORDER BY clause.
- It cannot refer to the table into which you are inserting rows.

The INTO, INTO TEMP, and ORDER BY clause restrictions are minor. The INTO clause is not useful in this context. (For more information, see SQL programming on page 1700.) To work around the INTO TEMP clause restriction, first select the data you want to insert into a temporary table and then insert the data from the temporary table with the INSERT statement. Likewise, the lack of an ORDER BY clause is not important. If you need to ensure that the new rows are physically ordered in

the table, you can first select them into a temporary table and order it, and then insert from the temporary table. You can also apply a physical order to the table using a clustered index after all insertions are done.

Important: The last restriction is more serious because it prevents you from naming the same table in both the INTO clause of the INSERT statement and the FROM clause of the SELECT statement. Naming the same table in both the INTO clause of the INSERT statement and the FROM clause of the SELECT statement causes the database server to enter an endless loop in which each inserted row is reselected and reinserted.

In some cases, however, you might want to select from the same table into which you must insert data. For example, suppose that you have learned that the Nikolus company supplies the same products as the Anza company, but at half the price. You want to add rows to the **stock** table to reflect the difference between the two companies. Optimally, you want to select data from all the Anza stock rows and reinsert it with the Nikolus manufacturer code. However, you cannot select from the same table into which you are inserting.

To get around this restriction, select the data you want to insert into a temporary table. Then select from that temporary table in the INSERT statement, as the following example shows:

```
SELECT stock_num, 'NIK' temp_manu, description, unit_price/2
      half_price, unit,  unit_descr FROM stock
   WHERE manu_code = 'ANZ'
      AND stock_num < 110
   INTO TEMP anzrows;

INSERT INTO stock SELECT * FROM anzrows;

DROP TABLE anzrows;
```

This SELECT statement takes existing rows from **stock** and substitutes a literal value for the manufacturer code and a computed value for the unit price. These rows are then saved in a temporary table, **anzrows**, which is immediately inserted into the **stock** table.

When you insert multiple rows, a risk exists that one of the rows contains invalid data that might cause the database server to report an error. When such an error occurs, the statement terminates early. Even if no error occurs, a small risk exists that a hardware or software failure might occur while the statement is executing (for example, the disk might fill up).

In either event, you cannot easily tell how many new rows were inserted. If you repeat the statement in its entirety, you might create duplicate rows, or you might not. Because the database is in an unknown state, you cannot know what to do. The solution lies in using transactions, as Interrupted modifications on page 1691 discusses.

Update rows

Use the UPDATE statement to change the contents of one or more existing rows of a table, according to the specifications of the SET clause. This statement takes two fundamentally different forms. One lets you assign specific values to columns by name; the other lets you assign a list of values (that might be returned by a SELECT statement) to a list of columns. In either case, if you are updating rows, and some of the columns have data integrity constraints, the data that you change must conform to the constraints placed on those columns. For more information, refer to Data integrity on page 1680.

Note: An alternative to the UPDATE statement is the MERGE statement, which can use the same SET clause syntax as the UPDATE statement to modify one or more values in existing rows of a table. The MERGE statement performs an outer join of a source table and a target table, and then updates rows in the target table with values from the result set of the join for which the join predicate evaluates to TRUE. Values in the source table are unchanged by the MERGE statement. Besides updating rows, the MERGE statement can optionally combine both UPDATE and INSERT operations, or can combine both DELETE and INSERT operations without updating any rows. For more information about the syntax and the restrictions on Update merges, Delete merges, and Insert merges, see the description of the MERGE statement in the *HCL® Informix® Guide to SQL: Syntax*.

Select rows to update

Either form of the UPDATE statement can end with a WHERE clause that determines which rows are modified. If you omit the WHERE clause, all rows are modified. To select the precise set of rows that need changing in the WHERE clause can be quite complicated. The only restriction on the WHERE clause is that the table that you update cannot be named in the FROM clause of a subquery.

The first form of an UPDATE statement uses a series of assignment clauses to specify new column values, as the following example shows:

```
UPDATE customer
   SET fname = 'Barnaby', lname = 'Dorfler'
   WHERE customer_num = 103;
```

The WHERE clause selects the row you want to update. In the demonstration database, the **customer.customer_num** column is the primary key for that table, so this statement can update no more than one row.

You can also use subqueries in the WHERE clause. Suppose that the Anza Corporation issues a safety recall of their tennis balls. As a result, any unshipped orders that include stock number 6 from manufacturer ANZ must be put on back order, as the following example shows:

```
UPDATE orders
   SET backlog = 'y'
   WHERE ship_date IS NULL
   AND order_num IN
      (SELECT DISTINCT items.order_num FROM items
         WHERE items.stock_num = 6
         AND items.manu_code = 'ANZ');
```

This subquery returns a column of order numbers (zero or more). The UPDATE operation then tests each row of **orders** against the list and performs the update if that row matches.

Update with uniform values

Each assignment after the keyword SET specifies a new value for a column. That value is applied uniformly to every row that you update. In the examples in the previous section, the new values were constants, but you can assign any expression, including one based on the column value itself. Suppose the manufacturer code HRO has raised all prices by five percent, and you must update the **stock** table to reflect this increase. Use the following statement:

```
UPDATE stock
   SET unit_price = unit_price * 1.05
   WHERE manu_code = 'HRO';
```

You can also use a subquery as part of the assigned value. When a subquery is used as an element of an expression, it must return exactly one value (one column and one row). Perhaps you decide that for any stock number, you must charge a higher price than any manufacturer of that product. You need to update the prices of all unshipped orders. The SELECT statements in the following example specify the criteria:

```
UPDATE items
   SET total_price = quantity *
      (SELECT MAX (unit_price) FROM stock
         WHERE stock.stock_num = items.stock_num)
   WHERE items.order_num IN
      (SELECT order_num FROM orders
         WHERE ship_date IS NULL);
```

The first SELECT statement returns a single value: the highest price in the **stock** table for a particular product. The first SELECT statement is a correlated subquery because, when a value from **items** appears in the WHERE clause for the first SELECT statement, you must execute the query for every row that you update.

The second SELECT statement produces a list of the order numbers of unshipped orders. It is an uncorrelated subquery that is executed once.

Restrictions on updates

Restrictions exist on the use of subqueries when you modify data. In particular, you cannot query the table that is being modified. You can refer to the present value of a column in an expression, as in the example that increments the **unit_price** column by 5 percent. You can also refer to a value of a column in a WHERE clause in a subquery, as in the example that updated the **stock** table, in which the **items** table is updated and **items.stock_num** is used in a join expression.

The need to update and query a table at the same time does not occur often in a well-designed database. (For more information about database design, see the *IBM® Informix® Database Design and Implementation Guide*.) However, you might want to update and query at the same time when a database is first being developed, before its design has been carefully thought through. A typical problem arises when a table inadvertently and incorrectly contains a few rows with duplicate values in a column that should be unique. You might want to delete the duplicate rows or update only the duplicate rows. Either way, a test for duplicate rows inevitably requires a subquery on the same table that you want to modify, which is not allowed in an UPDATE statement or DELETE statement. Modify data through SQL programs on page 1723 discusses how to use an *update cursor* to perform this kind of modification.

Update with selected values

The second form of UPDATE statement replaces the list of assignments with a single bulk assignment, in which a list of columns is set equal to a list of values. When the values are simple constants, this form is nothing more than the form of the previous example with its parts rearranged, as the following example shows:

```
UPDATE customer
   SET (fname, lname) = ('Barnaby', 'Dorfler')
   WHERE customer_num = 103;
```

No advantage exists to writing the statement this way. In fact, it is harder to read because it is not obvious which values are assigned to which columns.

However, when the values to be assigned come from a single SELECT statement, this form makes sense. Suppose that changes of address are to be applied to several customers. Instead of updating the **customer** table each time a change is reported, the new addresses are collected in a single temporary table named **newaddr**. It contains columns for the customer number and the address-related fields of the **customer** table. Now the time comes to apply all the new addresses at once.

```
UPDATE customer
  SET (address1, address2, city, state, zipcode) =
    ((SELECT address1, address2, city, state, zipcode
      FROM newaddr
      WHERE newaddr.customer_num=customer.customer_num))
  WHERE customer_num IN (SELECT customer_num FROM newaddr);
```

A single SELECT statement produces the values for multiple columns. If you rewrite this example in the other form, with an assignment for each updated column, you must write five SELECT statements, one for each column to be updated. Not only is such a statement harder to write, but it also takes much longer to execute.

Tip: In SQL API programs, you can use record or host variables to update values. For more information, refer to SQL programming on page 1700.

Update row types

The syntax you use to update a row-type value differs somewhat depending on whether the column is a named ROW type or unnamed ROW type. This section describes those differences and also describes how to specify NULL values for the fields of a ROW type.

Update rows that contain named row types

To update a column that is defined on a named ROW type, you must specify all fields of the ROW type. For example, the following statement updates only the **street** and **city** fields of the **address** column in the **employee** table, but each field of the ROW type must contain a value (NULL values are allowed):

```
UPDATE employee
  SET address = ROW('103 California St',
    San Francisco', address.state, address.zip)::address_t
  WHERE name = 'zawinul, joe';
```

In this example, the values of the **state** and **zip** fields are read from and then immediately reinserted into the row. Only the **street** and **city** fields of the **address** column are updated.

When you update the fields of a column that are defined on a named ROW type, you must use a ROW constructor and cast the row value to the appropriate named ROW type.

Update rows that contain unnamed row types

To update a column that is defined on an unnamed ROW type, you must specify all fields of the ROW type. For example, the following statement updates only the **street** and **city** fields of the **address** column in the **student** table, but each field of the ROW type must contain a value (NULL values are allowed):

```
UPDATE student
   SET s_address = ROW('13 Sunset', 'Fresno',
   s_address.state, s_address.zip)
   WHERE s_name = 'henry, john';
```

To update the fields of a column that are defined on an unnamed ROW type, always specify the ROW constructor before the field values to be inserted.

Specify Null values for the fields of a row type

The fields of a row-type column can contain NULL values. When you insert into or update a row-type field with a NULL value, you must cast the value to the data type of that field.

The following UPDATE statement shows how you might specify NULL values for particular fields of a named row-type column:

```
UPDATE employee
   SET address = ROW(NULL::VARCHAR(20), 'Davis', 'CA',
   ROW(NULL::CHAR(5), NULL::CHAR(4)))::address_t)
   WHERE name = 'henry, john';
```

The following UPDATE statement shows how you specify NULL values for the **street** and **zip** fields of the **address** column for the **student** table.

```
UPDATE student
   SET address = ROW(NULL::VARCHAR(20), address.city,
   address.state, NULL::VARCHAR(9))
   WHERE s_name = 'henry, john';
```

Important: You cannot specify NULL values for a row-type column. You can only specify NULL values for the individual fields of the row type.

Update collection types

When you use to update a collection type, you must update the entire collection. The following statement shows how to update the **projects** column. To locate the row that needs to be updated, use the IN keyword to perform a search on the **direct_reports** column.

```
UPDATE manager
SET projects = "LIST
{
   ROW('brazil_project', SET{'Pryor', 'Murphy', 'Kinsley',
      'Bryant'}),
   ROW ('cuba_project', SET{'Forester', 'Barth', 'Lewis',
      'Leonard'})
```

```
}"
WHERE 'Williams' IN direct_reports;
```

The first occurrence of the SET keyword in the preceding statement is part of the UPDATE statement syntax.

Important: Do not confuse the SET keyword of an UPDATE statement with the SET constructor that indicates that a collection is a SET data type.

Although you can use the IN keyword to locate specific elements of a simple collection, you cannot update individual elements of a collection column from . However, you can create programs and SPL routines to update elements within a collection. For information about how to create programs to update a collection, see the *HCL® Informix® Enterprise Replication Guide*. For information about how to create SPL routines to update a collection, see the section Handle collections on page 1786.

Update rows of a supertable

When you update the rows of a supertable, the scope of the update is a supertable and its subtables.

When you construct an UPDATE statement on a supertable, you can update all columns in the supertable and columns of subtables that are inherited from the supertable. For example, the following statement updates rows from the **employee** and **sales_rep** tables, which are subtables of the supertable **person**:

```
UPDATE person
   SET salary=65000
   WHERE address.state = 'CA';
```

However, an update on a supertable does not allow you to update columns from subtables that are not in the supertable. For example, in the previous update statement, you cannot update the **region_num** column of the **sales_rep** table because the **region_num** column does not occur in the **employee** table.

When you perform updates on supertables, be aware of the scope of the update. For example, an UPDATE statement on the **person** table that does not include a WHERE clause to restrict which rows to update, modifies all rows of the **person**, **employee**, and **sales_rep** table.

To limit an update to rows of the supertable only, you must use the ONLY keyword in the UPDATE statement. For example, the following statement updates rows of the **person** table only:

```
UPDATE ONLY(person)
   SET address = ROW('14 Jackson St', 'Berkeley',
   address.state, address.zip)
   WHERE name = 'Sallie, A.';
```

Important: Use caution when you update rows of a supertable because the scope of an update on a supertable includes the supertable and all its subtables.

CASE expression to update a column

The CASE expression allows a statement to return one of several possible results, depending on which of several condition tests evaluates to TRUE.

The following example shows how to use a CASE expression in an UPDATE statement to increase the unit price of certain items in the **stock** table:

```
UPDATE stock
   SET unit_price = CASE
      WHEN stock_num = 1
      AND manu_code = "HRO"
      THEN unit_price * 1.2
      WHEN stock_num = 1
      AND manu_code = "SMT"
      THEN unit_price * 1.1
      ELSE 0
      END
```

You must include at least one WHEN clause within the CASE expression; subsequent WHEN clauses and the ELSE clause are optional. If no WHEN condition evaluates to true, the resulting value is null.

SQL functions to update smart large objects

You can use an SQL function that you can call from within an UPDATE statement to import and export smart large objects. For a description of these functions, see page Smart large object functions on page 1603.

The following UPDATE statement uses the LOCOPY() function to copy BLOB data from the **mugshot** column of the **fbi_list** table into the **picture** column of the **inmate** table. (Figure 231: Create the inmate and fbi_list tables. on page 1604 defines the **inmate** and **fbi_list** tables.)

```
UPDATE inmate (picture)
    SET picture = (SELECT LOCOPY(mugshot, 'inmate', 'picture')
                   FROM fbi_list WHERE fbi_list.id = 669)
    WHERE inmate.id_num = 437;
```

The first argument for LOCOPY() specifies the column (**mugshot**) from which the object is exported. The second and third arguments specify the name of the table (**inmate**) and column (**picture**) whose storage characteristics the newly created object will use. After execution of the UPDATE statement, the **picture** column contains data from the **mugshot** column.

When you specify the path of a file name in the function argument, apply the following rules:

- If the source file resides on the server computer, you must specify the full path name to the file (not the path name relative to the current working directory).
- If the source file resides on the client computer, you can specify either the full or relative path name to the file.

The MERGE statement to update a table

The MERGE statement allows you to apply a Boolean condition to the result of an outer join of a source table and a target table. If the MERGE statement includes the Update clause, rows that satisfy the join condition that you specify after the ON keyword are used in UPDATE operations on the target. The SET clause of the MERGE statement supports the same syntax as the SET clause of the UPDATE statement, and specifies which columns of the target table to update.

The following example illustrates how you can use the Update clause of the MERGE statement to update a target table:

```
MERGE INTO t_target AS t USING t_source AS s ON t.col_a = s.col_a
   WHEN MATCHED THEN UPDATE
      SET t.col_b = t.col_b + s.col_b ;
```

In the preceding example, the name of the target table is **t_target** and the name of the source table is **t_source**. For rows of the join result where **col_a** has the same value in both the source and the target tables, the MERGE statement updates the **t_target** table by adding the value of column **col_b** in the source table to the current value of the **col_b** column in the **t_target** table.

An UPDATE operation of the MERGE statement does not modify the source table, and cannot update any row in the target table more than once.

A single MERGE statement can combine both UPDATE and INSERT operations, or can combine both DELETE and INSERT operations but the delete clause is not required. For a different example of MERGE that includes no Update clause, see the topic The Delete clause of MERGE on page 1662

Privileges on a database and on its objects

You can use the following database privileges to control who accesses a database:

- Database-level privileges
- Table-level privileges
- Routine-level privileges
- Language-level privileges
- Type-level privileges
- Sequence-level privileges
- Fragment-level privileges

This section briefly describes database- and table-level privileges. For more information about database privileges, see the *IBM® Informix® Database Design and Implementation Guide*. For a list of privileges and a description of the GRANT and REVOKE statements, see the *HCL® Informix® Guide to SQL: Syntax*.

Database-level privileges

When you create a database, you are the only one who can access it until you, as the owner or database administrator (DBA) of the database, grant database-level privileges to others. The following table shows database-level privileges.

Privilege	Effect
Connect	Allows you to open a database, issue queries, and create and place indexes on temporary tables.
Resource	Allows you to create permanent tables.
DBA	Allows you to perform several additional functions as the DBA.

Table-level privileges

When you create a table in a database that is not ANSI compliant, all users have access privileges to the table until you, as the owner of the table, revoke table-level privileges from specific users. The following table introduces the four privileges that govern how users can access a table.

Privilege	Purpose
Select	Granted on a table-by-table basis and allows you to select rows from a table. (This privilege can be limited to specific columns in a table.)
Delete	Allows you to delete rows.
Insert	Allows you to insert rows.
Update	Allows you to update existing rows (that is, to change their content).

The people who create databases and tables often grant the Connect and Select privileges to **public** so that all users have them. If you can query a table, you have at least the Connect and Select privileges for that database and table.

You need the other table-level privileges to modify data. The owners of tables often withhold these privileges or grant them only to specific users. As a result, you might not be able to modify some tables that you can query freely.

Because these privileges are granted on a table-by-table basis, you can have only Insert privileges on one table and only Update privileges on another, for example. The Update privileges can be restricted even further to specific columns in a table.

For more information on these and other table-level privileges, see the *IBM® Informix® Database Design and Implementation Guide*.

Display table privileges

If you are the owner of a table (that is, if you created it), you have all privileges on that table. Otherwise, you can determine the privileges you have for a certain table by querying the system catalog. The system catalog consists of system tables that describe the database structure. The privileges granted on each table are recorded in the **systabauth** system table. To display these privileges, you must also know the unique identifier number of the table. This number is specified in the **systables** system table. To display privileges granted on the **orders** table, you might enter the following SELECT statement:

```
SELECT * FROM systabauth
  WHERE tabid = (SELECT tabid FROM systables
                      WHERE tabname = 'orders');
```

The output of the query resembles the following example:

```
grantorgrantee tabid              tabauth

tfecitmutator  101      su-i-x--
tfecitprocrustes101     s--idx--
tfecitpublic   101      s--i-x--
```

The grantor is the user who grants the privilege. The grantor is usually the owner of the table but the owner can be another user that the grantor empowered. The grantee is the user to whom the privilege is granted, and the grantee **public** means any user with Connect privilege. If your user name does not appear, you have only those privileges granted to **public**.

The **tabauth** column specifies the privileges granted. The letters in each row of this column are the initial letters of the privilege names, except that `i` means Insert and `x` means Index. In this example, **public** has Select, Insert, and Index privileges. Only the user **mutator** has Update privileges, and only the user **procrustes** has Delete privileges.

Before the database server performs any action for you (for example, execution of a DELETE statement), it performs a query similar to the preceding one. If you are not the owner of the table, and if the database server cannot find the necessary privilege on the table for your user name or for **public**, it refuses to perform the operation.

Grant privileges to roles

As DBA, you can create roles to standardize the privileges given to a class of users. When you assign privileges to that role, every user of that role has those access privileges. The SQL statements used for defining and manipulating roles include: CREATE ROLE, DROP ROLE, GRANT, REVOKE, and SET ROLE. For more information on the SQL syntax statements for defining and manipulating roles, see the *HCL® Informix® Guide to SQL: Syntax*.

Default roles automatically apply upon connection to the database for particular users and groups, without requiring the user to issue a SET ROLE statement. For example:

```
GRANT DEFAULT ROLE manager TO larry;
```

For more information on roles and default roles, see Control database use on page 1524 or see the *HCL® Informix® Administrator's Guide*.

For more information on granting and revoking privileges, see Grant and revoke privileges in applications on page 1720. Also see *IBM® Informix® Database Design and Implementation Guide*.

Data integrity

The INSERT, UPDATE, and DELETE statements modify data in an existing database. Whenever you modify existing data, the *integrity* of the data can be affected. For example, an order for a nonexistent product could be entered into the **orders** table, a customer with outstanding orders could be deleted from the **customer** table, or the order number could be updated in the **orders** table and not in the **items** table. In each of these cases, the integrity of the stored data is lost.

Data integrity is actually made up of the following parts:

Entity integrity

Each row of a table has a unique identifier.

Semantic integrity

The data in the columns properly reflects the types of information the column was designed to hold.

Referential integrity

The relationships between tables are enforced.

Well-designed databases incorporate these principles so that when you modify data, the database itself prevents you from doing anything that might harm the integrity of the data.

Entity integrity

An entity is any person, place, or thing to be recorded in a database. Each table represents an entity, and each row of a table represents an instance of that entity. For example, if *order* is an entity, the **orders** table represents the idea of an order and each row in the table represents a specific order.

To identify each row in a table, the table must have a primary key. The primary key is a unique value that identifies each row. This requirement is called the *entity integrity constraint*.

For example, the **orders** table primary key is **order_num**. The **order_num** column holds a unique system-generated order number for each row in the table. To access a row of data in the **orders** table, use the following SELECT statement:

```
SELECT * FROM orders WHERE order_num = 1001;
```

Using the order number in the WHERE clause of this statement enables you to access a row easily because the order number uniquely identifies that row. If the table allowed duplicate order numbers, it would be almost impossible to access one single row because all other columns of this table allow duplicate values.

For more information on primary keys and entity integrity, see the *IBM® Informix® Database Design and Implementation Guide*.

Semantic integrity

Semantic integrity ensures that data entered into a row reflects an allowable value for that row. The value must be within the *domain*, or allowable set of values, for that column. For example, the **quantity** column of the **items** table permits only numbers. If a value outside the domain can be entered into a column, the semantic integrity of the data is violated.

The following constraints enforce semantic integrity:

Data type

The data type defines the types of values that you can store in a column. For example, the data type SMALLINT allows you to enter values from -32,767 to 32,767 into a column.

Default value

The default value is the value inserted into the column when an explicit value is not specified. For example, the **user_id** column of the **cust_calls** table defaults to the login name of the user if no name is entered.

Check constraint

The check constraint specifies conditions on data inserted into a column. Each row inserted into a table must meet these conditions. For example, the **quantity** column of the **items** table might check for quantities greater than or equal to one.

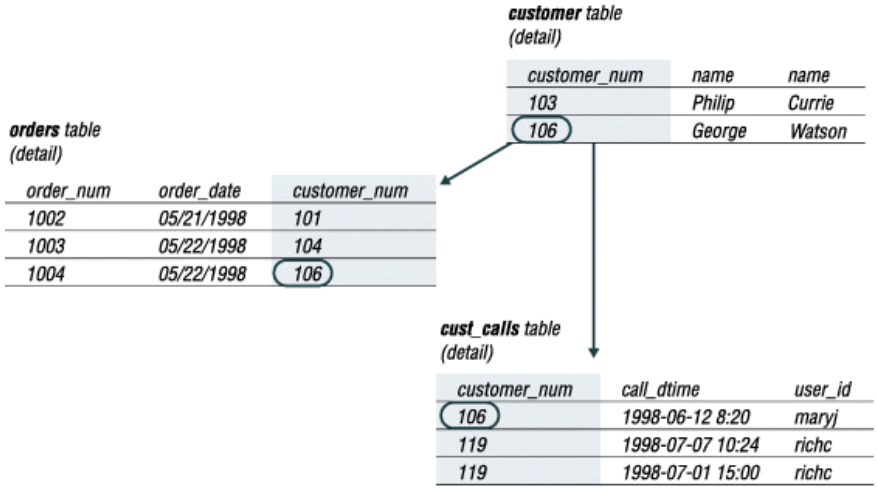
For more information on how to use semantic integrity constraints in database design, see the *IBM® Informix® Database Design and Implementation Guide*.

Referential integrity

Referential integrity refers to the relationship between tables. Because each table in a database must have a primary key, this primary key can appear in other tables because of its relationship to data within those tables. When a primary key from one table appears in another table, it is called a *foreign key*.

Foreign keys join tables and establish dependencies between tables. tables can form a hierarchy of dependencies in such a way that if you change or delete a row in one table, you destroy the meaning of rows in other tables. For example, the following figure shows that the **customer_num** column of the **customer** table is a primary key for that table and a foreign key in the **orders** and **cust_call** tables. Customer number 106, George Watson™, is *referenced* in both the **orders** and **cust_calls** tables. If customer 106 is deleted from the **customer** table, the link between the three tables and this particular customer is destroyed.

Figure 379. Referential integrity in the demonstration database



When you delete a row that contains a primary key or update it with a different primary key, you destroy the meaning of any rows that contain that value as a foreign key. Referential integrity is the logical dependency of a foreign key on a primary key. The integrity of a row that contains a foreign key depends on the integrity of the row that it references-the row that contains the matching primary key.

By default, the database server does not allow you to violate referential integrity and gives you an error message if you attempt to delete rows from the parent table before you delete rows from the child table. You can, however, use the ON DELETE CASCADE option to cause deletes from a parent table to trip deletes on child tables. See The ON DELETE CASCADE option on page 1683.

To define primary and foreign keys, and the relationship between them, use the CREATE TABLE and ALTER TABLE statements. For more information on these statements, see the *HCL® Informix® Guide to SQL: Syntax*. For information about how to build a data model with primary and foreign keys, see the *IBM® Informix® Database Design and Implementation Guide*.

The ON DELETE CASCADE option

To maintain referential integrity when you delete rows from a primary key for a table, use the ON DELETE CASCADE option in the REFERENCES clause of the CREATE TABLE and ALTER TABLE statements. This option allows you to delete a row from a parent table and its corresponding rows in matching child tables with a single delete command.

Lock during cascading deletes

During deletes, locks are held on all qualifying rows of the parent and child tables. When you specify a delete, the delete that is requested from the parent table occurs before any referential actions are performed.

What happens to multiple children tables

If you have a parent table with two child constraints, one child with cascading deletes specified and one child without cascading deletes, and you attempt to delete a row from the parent table that applies to both child tables, the DELETE statement fails, and no rows are deleted from either the parent or child tables.

Logging must be turned on

You must turn on logging in your current database for cascading deletes to work. Logging and cascading deletes are discussed in Transaction logging on page 1692.

Example of cascading deletes

Suppose you have two tables with referential integrity rules applied, a parent table, accounts, and a child table, **sub_accounts**. The following CREATE TABLE statements define the referential constraints:

```
CREATE TABLE accounts (
  acc_num SERIAL primary key,
  acc_type INT,
  acc_descr CHAR(20));

CREATE TABLE sub_accounts (
  sub_acc INTEGER primary key,
  ref_num INTEGER REFERENCES accounts (acc_num)
    ON DELETE CASCADE,
  sub_descr CHAR(20));
```

The primary key of the accounts table, the **acc_num** column, uses a SERIAL data type, and the foreign key of the **sub_accounts** table, the **ref_num** column, uses an INTEGER data type. Combining the SERIAL data type on the primary key and the INTEGER data type on the foreign key is allowed. Only in this condition can you mix and match data types. The SERIAL data type is an INTEGER, and the database automatically generates the values for the column. All other primary and foreign key combinations must match explicitly. For example, a primary key that is defined as CHAR must match a foreign key that is defined as CHAR.

The definition of the foreign key of the **sub_accounts** table, the **ref_num** column, includes the ON DELETE CASCADE option. This option specifies that a delete of any row in the parent table **accounts** will automatically cause the corresponding rows of the child table **sub_accounts** to be deleted.

To delete a row from the accounts table that will cascade a delete to the **sub_accounts** table, you must turn on logging. After logging is turned on, you can delete the account number 2 from both tables, as the following example shows:

```
DELETE FROM accounts WHERE acc_num = 2;
```

Restrictions on cascading deletes

You can use cascading deletes for most deletes, including deletes on self-referencing and cyclic queries. The only exception is correlated subqueries, which are nested SELECT statements in which the value that the subquery (or inner SELECT) produces depends on a value produced by the outer SELECT statement that contains it. If you have implemented cascading deletes, you cannot write deletes that use a child table in the correlated subquery. You receive an error when you attempt to delete from a correlated subquery.

Restriction: You cannot define a DELETE trigger event on a table if the table defines a referential constraint with ON DELETE CASCADE.

Object modes and violation detection

The object modes and violation detection features of the database can help you monitor data integrity. These features are particularly powerful when they are combined during schema changes or when insert, delete, and update operations are performed on large volumes of data over short periods.

Database objects, within the context of a discussion of the object modes feature, are constraints, indexes, and triggers, and each of them have different modes. Do not confuse database objects that are relevant to the object modes feature with generic database objects. Generic database objects are things like tables and synonyms.

Definitions of object modes

You can set disabled, enabled, or filtering modes for a constraint or a unique index. You can set disabled or enabled modes for a trigger or a duplicate index. You can use database object modes to control the effects of INSERT, DELETE, and UPDATE statements.

Enabled mode

Constraints, indexes, and triggers are enabled by default.

When a database object is enabled, the database server recognizes the existence of the database object and takes the database object into consideration while it executes an INSERT, DELETE, or UPDATE statement. Thus, an enabled constraint is enforced, an enabled index updated, and an enabled trigger is executed when the trigger event takes place.

When you enable constraints and unique indexes, if a violating row exists, the data manipulation statement fails (that is no rows change) and the database server returns an error message.

You can identify the reason for the failure when you analyze the information in the violations and diagnostic tables. You can then take corrective action or roll back the operation.

Disabled mode

When a database object is disabled, the database server does not take it into consideration during the execution of an INSERT, DELETE, or UPDATE statement. A disabled constraint is not enforced, a disabled index is not updated, and a disabled trigger is not executed when the trigger event takes place. When you disable constraints and unique indexes, any data manipulation statement that violates the restriction of the constraint or unique index succeed, (that is, the target row is changed), and the database server does not return an error message.

Filtering mode

When a constraint or unique index is in filtering mode, the statement succeeds and the database server enforces the constraint or the unique index requirement during an INSERT, DELETE, or UPDATE statement by writing the failed rows to the violations table associated with the target table. Diagnostic information about the constraint violation is written to the diagnostics table associated with the target table.

Example of modes with data manipulation statements

An example with the INSERT statement can illustrate the differences between the enabled, disabled, and filtering modes. Consider an INSERT statement in which a user tries to add a row that does not satisfy an integrity constraint on a table. For example, assume that user **joe** created a table named **cust_subset**, and this table consists of the following columns: **ssn** (customer's social security number), **fname** (customer's first name), **lname** (customer's last name), and **city** (city in which the customer lives). The **ssn** column has the INT data type. The other three columns have the CHAR data type.

Assume that user **joe** defined the **lname** column as not null but has not assigned a name to the not null constraint, so the database server has implicitly assigned the name **n104_7** to this constraint. Finally, assume that user **joe** created a unique index named **unq_ssn** on the **ssn** column.

Now user **linda** who has the Insert privilege on the **cust_subset** table enters the following INSERT statement on this table:

```
INSERT INTO cust_subset (ssn, fname, city)
   VALUES (973824499, "jane", "los altos");
```

To better understand the distinctions among enabled, disabled, and filtering modes, you can view the results of the preceding INSERT statement in the following three sections.

Results of the insert operation when the constraint is enabled

If the NOT NULL constraint on the **cust_subset** table is enabled, the INSERT statement fails to insert the new row in this table. Instead user **linda** receives the following error message when they enter the INSERT statement:

```
-292 An implied insert column lname does not accept NULLs.
```

Results of the insert operation when the constraint is disabled

If the NOT NULL constraint on the **cust_subset** table is disabled, the INSERT statement that user **linda** issues successfully inserts the new row in this table. The new row of the **cust_subset** table has the following column values.

ssn	fname	lname	city
973824499	jane	NULL	los altos

Results of the insert when constraint is in filtering mode

If the NOT NULL constraint on the **cust_subset** table is set to the filtering mode, the INSERT statement that user **linda** issues fails to insert the new row in this table. Instead the new row is inserted into the violations table, and a diagnostic row that describes the integrity violation is added to the diagnostics table.

Assume that user **joe** has started a violations and diagnostics table for the **cust_subset** table. The violations table is named **cust_subset_vio**, and the diagnostics table is named **cust_subset_dia**. The new row added to the **cust_subset_vio** violations table when user **linda** issues the INSERT statement on the **cust_subset** target table has the following column values.

ssn	fname	lname	city	informix_tupleid	informix_optype	informix_recowner
973824499	jane	NULL	los altos	1	I	linda

This new row in the **cust_subset_vio** violations table has the following characteristics:

- The first four columns of the violations table exactly match the columns of the target table. These four columns have the same names and the same data types as the corresponding columns of the target table, and they have the column values that were supplied by the INSERT statement that user **linda** entered.
- The value `1` in the **informix_tupleid** column is a unique serial identifier that is assigned to the nonconforming row.
- The value `I` in the **informix_optype** column is a code that identifies the type of operation that has caused this nonconforming row to be created. Specifically, `I` stands for an INSERT operation.
- The value `linda` in the **informix_recowner** column identifies the user who issued the statement that caused this nonconforming row to be created.

The INSERT statement that user **linda** issued on the **cust_subset** target table also causes a diagnostic row to be added to the **cust_subset_dia** diagnostics table. The new diagnostic row added to the diagnostics table has the following column values.

informix_tupleid	objtype	objowner	objname
1	C	joe	n104_7

This new diagnostic row in the **cust_subset_dia** diagnostics table has the following characteristics:

- This row of the diagnostics table is linked to the corresponding row of the violations table by means of the **informix_tupleid** column that appears in both tables. The value `1` appears in this column in both tables.
- The value `C` in the **objtype** column identifies the type of integrity violation that the corresponding row in the violations table caused. Specifically, the value `C` stands for a constraint violation.
- The value `joe` in the **objowner** column identifies the owner of the constraint for which an integrity violation was detected.
- The value `n104_7` in the **objname** column gives the name of the constraint for which an integrity violation was detected.

By joining the violations and diagnostics tables, user **joe** (who owns the **cust_subset** target table and its associated special tables) or the DBA can find out that the row in the violations table whose **informix_tupleid** value is `1` was created after an INSERT statement and that this row is violating a constraint. The table owner or DBA can query the **sysconstraints** system catalog table to determine that this constraint is a NOT NULL constraint. Now that the reason for the failure of the INSERT statement is known, user **joe** or the DBA can take corrective action.

Multiple diagnostic rows for one violations row

In the preceding example, only one row in the diagnostics table corresponds to the new row in the violations table. However, more than one diagnostic row can be added to the diagnostics table when a single new row is added to the violations table. For example, if the **ssn** value (973824499) that user **linda** entered in the INSERT statement had been the same as an existing value in the **ssn** column of the **cust_subset** target table, only one new row would appear in the violations table, but the following two diagnostic rows would be present in the **cust_subset_dia** diagnostics table.

informix_tupleid	objtype	objowner	objname
1	C	joe	n104_7
1	I	joe	unq_ssn

Both rows in the diagnostics table correspond to the same row of the violations table because both of these rows have the value `1` in the **informix_tupleid** column. The first diagnostic row, however, identifies the constraint violation caused by the INSERT statement that user **linda** issued, while the second diagnostic row identifies the unique-index violation that the same INSERT statement caused. In this second diagnostic row, the value `I` in the **objtype** column stands for a unique-index violation, and the value `unq_ssn` in the **objname** column gives the name of the index for which the integrity violation was detected.

For more information about how to set database object modes, see the SET Database Object Mode statement in the *HCL® Informix® Guide to SQL: Syntax*.

Violations and diagnostics tables

When you start a violations table for a target table, any rows that violate constraints and unique indexes during INSERT, UPDATE, and DELETE operations on the target table do not cause the entire operation to fail, but are filtered out to the violations table. The diagnostics table contains information about the integrity violations caused by each row in the violations table. By examining these tables, you can identify the cause of failure and take corrective action by either fixing the violation or rolling back the operation.

After you create a violations table for a target table, you cannot alter the columns or the fragmentation of the base table or the violations table. If you alter the constraints on a target table after you have started the violations table, nonconforming rows will be filtered to the violations table.

For information about how to start and stop the violations tables, see the START VIOLATIONS TABLE and STOP VIOLATIONS TABLE statements in the *HCL® Informix® Guide to SQL: Syntax*.

Relationship of violations tables and database object modes

If you set the constraints or unique indexes defined on a table to the filtering mode, but you do not create the violations and diagnostics tables for this target table, any rows that violate a constraint or unique-index requirement during an insert, update, or delete operation are not filtered to a violations table. Instead, you receive an error message that indicates that you must start a violations table for the target table.

Similarly, if you set a disabled constraint or disabled unique index to the enabled or filtering mode and you want the ability to identify existing rows that do not satisfy the constraint or unique-index requirement, you must create the violations tables before you issue the SET Database Object Mode statement.

Examples of START VIOLATIONS TABLE statements

The examples that follow show different ways to execute the START VIOLATIONS TABLE statement.

Start violations and diagnostics tables without specifying their names

To start a violations and diagnostics table for the target table named **customer** in the demonstration database, enter the following statement:

```
START VIOLATIONS TABLE FOR customer;
```

Because your START VIOLATIONS TABLE statement does not include a USING clause, the violations table is named **customer_vio** by default, and the diagnostics table is named **customer_dia** by default. The **customer_vio** table includes the following columns:

```
customer_num
fname
lname
company
address1
address2
city
state
zipcode
```

```
phone
informix_tupleid
informix_optype
informix_recowner
```

The **customer_vio** table has the same table definition as the **customer** table except that the **customer_vio** table has three additional columns that contain information about the operation that caused the bad row.

The **customer_dia** table includes the following columns:

```
informix_tupleid
objtype
objowner
objname
```

This list of columns shows an important difference between the diagnostics table and violations table for a target table. Whereas the violations table has a matching column for every column in the target table, the columns of the diagnostics table are independent of the schema of the target table. The diagnostics table created by any START VIOLATIONS TABLE statement always has the four columns in the list above, with the same column names and data types.

Start violations and diagnostics tables and specify their names

The following statement starts a violations and diagnostics table for the target table named **items**. The USING clause declares explicit names for the violations and diagnostics tables. The violations table is to be named **exceptions**, and the diagnostics table is to be named **reasons**.

```
START VIOLATIONS TABLE FOR items
   USING exceptions, reasons;
```

Specify the maximum number of rows in the diagnostics table

The following statement starts violations and diagnostics tables for the target table named **orders**. The MAX ROWS clause specifies the maximum number of rows that can be inserted into the **orders_dia**diagnostics table when a single statement, such as an INSERT, MERGE, or SET Database Object Mode, is executed on the target table.

```
START VIOLATIONS TABLE FOR orders MAX ROWS 50000;
```

If you do not specify a value for MAX ROWS in the START VIOLATIONS TABLE statement, there is no default limit on the number of rows in the diagnostics table, apart from the available disk space.

The MAX ROWS clause limits the number of rows only for operations in which the table functions as a diagnostics table.

Example of privileges on the violations table

The following example illustrates how the initial set of privileges on a violations table is derived from the current set of privileges on the target table.

For example, assume that we created a table named **cust_subset** and that this table consists of the following columns: **ssn** (customer's social security number), **fname** (customer's first name), **lname** (customer's last name), and **city** (city in which the customer lives).

The following set of privileges exists on the **cust_subset** table:

- User **alvin** is the owner of the table.
- User **barbara** has the Insert and Index privileges on the table. She also has the Select privilege on the **ssn** and **lname** columns.
- User **carrie** has the Update privilege on the **city** column. She also has the Select privilege on the **ssn** column.
- User **danny** has the Alter privilege on the table.

Now user **alvin** starts a violations table named **cust_subset_viols** and a diagnostics table named **cust_subset_diags** for the **cust_subset** table, as follows:

```
START VIOLATIONS TABLE FOR cust_subset
   USING cust_subset_viols, cust_subset_diags;
```

The database server grants the following set of initial privileges on the **cust_subset_viols** violations table:

- User **alvin** is the owner of the violations table, so he has all table-level privileges on the table.
- User **barbara** has the Insert, Delete, and Index privileges on the violations table. She also has the Select privilege on the following columns of the violations table: the **ssn** column, the **lname** column, the **informix_tupleid** column, the **informix_optype** column, and the **informix_recowner** column.
- User **carrie** has the Insert and Delete privileges on the violations table. She has the Update privilege on the following columns of the violations table: the **city** column, the **informix_tupleid** column, the **informix_optype** column, and the **informix_recowner** column. She has the Select privilege on the following columns of the violations table: the **ssn** column, the **informix_tupleid** column, the **informix_optype** column, and the **informix_recowner** column.
- User **danny** has no privileges on the violations table.

Example of privileges on the diagnostics table

The following example illustrates how the initial set of privileges on a diagnostics table is derived from the current set of privileges on the target table.

For example, assume that a table called **cust_subset** consists of the following columns: **ssn** (customer's social security number), **fname** (customer's first name), **lname** (customer's last name), and **city** (city in which the customer lives).

The following set of privileges exists on the **cust_subset** table:

- User **alvin** is the owner of the table.
- User **barbara** has the Insert and Index privileges on the table. They also have the Select privilege on the **ssn** and **lname** columns.
- User **carrie** has the Update privilege on the **city** column. They also have the Select privilege on the **ssn** column.
- User **danny** has the Alter privilege on the table.

Now user **alvin** starts a violations table named **cust_subset_viols** and a diagnostics table named **cust_subset_diags** for the **cust_subset** table, as follows:

```
START VIOLATIONS TABLE FOR cust_subset
   USING cust_subset_viols, cust_subset_diags;
```

The database server grants the following set of initial privileges on the **cust_subset_diags** diagnostics table:

- User **alvin** is the owner of the diagnostics table, so he has all table-level privileges on the table.
- User **barbara** has the Insert, Delete, Select, and Index privileges on the diagnostics table.
- User **carrie** has the Insert, Delete, Select, and Update privileges on the diagnostics table.
- User **danny** has no privileges on the diagnostics table.

Interrupted modifications

Even if all the software is error-free and all the hardware is utterly reliable, the world outside the computer can interfere. Lightning might strike the building, interrupting the electrical supply and stopping the computer in the middle of your UPDATE statement. A more likely scenario occurs when a disk fills up or a user supplies incorrect data, causing your multirow insert to stop early with an error. In any case, whenever you modify data, you must assume that some unforeseen event can interrupt the modification.

When an external cause interrupts a modification, you cannot be sure how much of the operation was completed. Even in a single-row operation, you cannot know whether the data reached the disk or the indexes were properly updated.

If multirow modifications are a problem, multistatement modifications are worse. They are usually embedded in programs so you do not see the individual SQL statements being executed. For example, to enter a new order in the demonstration database, perform the following steps:

1. Insert a row in the **orders** table. (This insert generates an order number.)
2. For each item ordered, insert a row in the **items** table.

Two ways to program an order-entry application exist. One way is to make it completely interactive so that the program inserts the first row immediately and then inserts each item as the user enters data. But this approach exposes the operation to the possibility of many more unforeseen events: the customer's telephone disconnecting, the user pressing the wrong key, the user's terminal or computer losing power, and so on.

The following list describes the correct way to build an order-entry application:

- Accept all the data interactively.
- Validate the data, and expand it (look up codes in **stock** and **manufact**, for example).
- Display the information on the screen for inspection.
- Wait for the operator to make a final commitment.
- Perform the insertions quickly.

Even with these steps, an unforeseen circumstance can halt the program after it inserts the order but before it finishes inserting the items. If that happens, the database is in an unpredictable condition: its data integrity is compromised.

Transactions

The solution to all these potential problems is called the *transaction*. A transaction is a sequence of modifications that must be accomplished either completely or not at all. The database server guarantees that operations performed within the

bounds of a transaction are either completely and perfectly committed to disk, or the database is restored to the same state as before the transaction started.

The transaction is not merely protection against unforeseen failures; it also offers a program a way to escape when the program detects a logical error.

Transaction logging

The database server can keep a record of each change that it makes to the database during a transaction. If something happens to cancel the transaction, the database server automatically uses the records to reverse the changes. Many things can make a transaction fail. For example, the program that issues the SQL statements can fail or be terminated. As soon as the database server discovers that the transaction failed, which might be only after the computer and the database server are restarted, it uses the records from the transaction to return the database to the same state as before.

The process of keeping records of transactions is called *transaction logging* or simply *logging*. The records of the transactions, called *log records*, are stored in a portion of disk space separate from the database. This space is called the *logical log* because the log records represent logical units of the transactions.

HCL Informix® provides support to:

- Create nonlogging (raw) or logging (standard) tables in a logging database.
- Alter a table from nonlogging to logging and vice-versa using the ALTER TABLE statement.

supports nonlogging tables for fast loads of very large tables. It is recommended that you do not use nonlogging tables within a transaction. To avoid concurrency problems, use the ALTER TABLE statement to make the table standard (that is, logging) before you use the table in a transaction.

For more information about nonlogging tables for , see the *HCL® Informix® Administrator's Guide*. For the performance advantages of nonlogging tables, see the *HCL® Informix® Performance Guide*. For information about the ALTER TABLE statement, see the *HCL® Informix® Guide to SQL: Syntax*.

Most HCL® Informix® databases do not generate transaction records automatically. The DBA decides whether to make a database use transaction logging. Without transaction logging, you cannot roll back transactions.

Logging and cascading deletes

Logging must be turned on in your database for cascading deletes to work because, when you specify a cascading delete, the delete is first performed on the primary key of the parent table. If the system fails after the rows of the primary key of the parent table are performed but before the rows of the foreign key of the child table are deleted, referential integrity is violated. If logging is turned off, even temporarily, deletes do not cascade. After logging is turned back on, however, deletes can cascade again.

HCL Informix® allows you to turn on logging with the WITH LOG clause in the CREATE DATABASE statement.

Specify transactions

You can use two methods to specify the boundaries of transactions with SQL statements. In the most common method, you specify the start of a multistatement transaction by executing the BEGIN WORK statement. In databases that are created with the MODE ANSI option, no need exists to mark the beginning of a transaction. One is always in effect; you indicate only the end of each transaction.

In both methods, to specify the end of a successful transaction, execute the COMMIT WORK statement. This statement tells the database server that you reached the end of a series of statements that must succeed together. The database server does whatever is necessary to make sure that all modifications are properly completed and committed to disk.

A program can also cancel a transaction deliberately by executing the ROLLBACK WORK statement. This statement asks the database server to cancel the current transaction and undo any changes.

An order-entry application can use a transaction in the following ways when it creates a new order:

- Accept all data interactively
- Validate and expand it
- Wait for the operator to make a final commitment
- Execute BEGIN WORK
- Insert rows in the **orders** and **items** tables, checking the error code that the database server returns
- If no errors occurred, execute COMMIT WORK; otherwise execute ROLLBACK WORK

If any external failure prevents the transaction from being completed, the partial transaction rolls back when the system restarts. In all cases, the database is in a predictable state. Either the new order is completely entered, or it is not entered at all.

Backups and logs with HCL Informix® database servers

By using transactions, you can ensure that the database is always in a consistent state and that your modifications are properly recorded on disk. But the disk itself is not perfectly safe. It is vulnerable to mechanical failures and to flood, fire, and earthquake. The only safeguard is to keep multiple copies of the data. These redundant copies are called *backup copies*.

The *transaction log* (also called the *logical log*) complements the backup copy of a database. Its contents are a history of all modifications that occurred since the last time the database was backed up. If you ever need to restore the database from the backup copy, you can use the transaction log to roll the database forward to its most recent state.

The database server contains elaborate features to support backups and logging. Your database server archive and backup guide describes these features.

The database server has stringent requirements for performance and reliability (for example, it supports making backup copies while databases are in use).

The database server manages its own disk space, which is devoted to logging.

The database server performs logging concurrently for all databases using a limited set of log files. The log files can be copied to another medium (backed up) while transactions are active.

Database users never have to be concerned with these facilities because the DBA usually manages them from a central location.

HCL Informix® supports the onload and onunload utilities. Use the onunload utility to make a personal backup copy of a single database or table. This program copies a table or a database to tape. Its output consists of binary images of the disk pages as they were stored in the database server. As a result, the copy can be made quickly, and the corresponding onload program can restore the file quickly. However, the data format is not meaningful to any other programs. For information about how to use the onload and onunload utilities, see the *HCL Informix® Migration Guide*.

If your DBA uses to create backups and back up logical logs, you might also be able to create your own backup copies using . For more information, see your *HCL® Informix® Backup and Restore Guide*.

Concurrency and locks

If your database is contained in a single-user workstation, without a network connecting it to other computers, concurrency is unimportant. In all other cases, you must allow for the possibility that, while your program is modifying data, another program is also reading or modifying the same data. *Concurrency* involves two or more independent uses of the same data at the same time.

A high level of concurrency is crucial to good performance in a multiuser database system. Unless controls exist on the use of data, however, concurrency can lead to a variety of negative effects. Programs could read obsolete data; modifications could be lost even though it seems they were entered successfully.

To prevent errors of this kind, the database server imposes a system of *locks*. A lock is a claim, or reservation, that a program can place on a piece of data. The database server guarantees that, as long as the data is locked, no other program can modify it. When another program requests the data, the database server either makes the program wait or turns it back with an error.

To control the effect that locks have on your data access, use a combination of SQL statements: SET LOCK MODE and either SET ISOLATION or SET TRANSACTION. You can understand the details of these statements after reading a discussion on the use of *cursors* from within programs. Cursors are covered in SQL programming on page 1700, and Modify data through SQL programs on page 1723. For more information about locking and concurrency, see Programming for a multiuser environment on page 1733.

HCL Informix® data replication

Data replication, in the broadest sense of the term, means that database objects have more than one representation at more than one distinct site. For example, one way to replicate data, so that reports can be run against the data without disturbing client applications that are using the original database, is to copy the database to a database server on a different computer.

The following list describes the advantages of data replication:

- Clients who access replicated data locally, as opposed to remote data that is not replicated, experience improved performance because they do not have to use network services.
- Clients at all sites experience improved availability with replicated data, because if local replicated data is unavailable, a copy of the data is still available, albeit remotely.

These advantages do not come without a cost. Data replication obviously requires more storage for replicated data than for unreplicated data, and updating replicated data can take more processing time than updating a single object.

Data replication can actually be implemented in the logic of client applications, by explicitly specifying where data should be found or updated. However, this method of achieving data replication is costly, error-prone, and difficult to maintain. Instead, the concept of data replication is often coupled with *replication transparency*. Replication transparency is functionality built into a database server (instead of client applications) to handle the details of locating and maintaining data replicas automatically.

Within the broad framework of data replication, a database server implements nearly transparent data replication of entire database servers. All the data that one database server manages is replicated and dynamically updated on another database server, usually at a remote site. Data replication of the HCL Informix® database server is sometimes called *hot-site backup*, because it provides a means of maintaining a backup copy of the entire database server that can be used quickly in the event of a catastrophic failure.

Because the database server provides replication transparency, you generally do not need to be concerned with or aware of data replication; the DBA takes care of it. However, if your organization decides to use data replication, you should be aware that special connectivity considerations exist for client applications in a data replication environment. These considerations are described in the *HCL® Informix® Administrator's Guide*.

The HCL Informix® feature provides a different method of data replication. For information on this feature, see the HCL Informix® Enterprise Replication Guide.

Summary

Database access is regulated by the privileges that the database owner grants to you. The privileges that let you query data are often granted automatically, but the ability to modify data is regulated by specific Insert, Delete, and Update privileges that are granted on a table-by-table basis.

If data integrity constraints are imposed on the database, your ability to modify data is restricted by those constraints. Your database- and table-level privileges and any data constraints control how and when you can modify data. In addition, the object modes and violation detection features of the database affect how you can modify data and help to preserve the integrity of your data.

You can delete one or more rows from a table with the DELETE statement. Its WHERE clause selects the rows; use a SELECT statement with the same clause to preview the deletes.

The TRUNCATE statement deletes all the rows of a table.

Rows are added to a table with the INSERT statement. You can insert a single row that contains specified column values, or you can insert a block of rows that a SELECT statement generates.

Use the UPDATE statement to modify the contents of existing rows. You specify the new contents with expressions that can include subqueries, so that you can use data that is based on other tables or the updated table itself. The statement has two forms. In the first form, you specify new values column by column. In the second form, a SELECT statement or a record variable generates a set of new values.

Use the REFERENCES clause of the CREATE TABLE and ALTER TABLE statements to create relationships between tables. The ON DELETE CASCADE option of the REFERENCES clause allows you to delete rows from parent and associated child tables with one DELETE statement.

Use transactions to prevent unforeseen interruptions in a modification from leaving the database in an indeterminate state. When modifications are performed within a transaction, they are rolled back after an error occurs. The transaction log also extends the periodically made backup copy of the database. If the database must be restored, it can be brought back to its most recent state.

Data replication, which is transparent to users, offers another type of protection from catastrophic failures.

Access and modify data in an external database

This section summarizes accessing tables and routines that are not in the current database.

Access other database servers

You can access any table or routine in an *external* database by qualifying the name of the database object (table, view, synonym, or routine).

When the external database is on the same database server as the current database, you must qualify the object name with the database name and a colon. For example, to refer to a table in a database other than the local database, the following SELECT statement accesses information from an external database:

```
SELECT name, number FROM salesdb:contacts
```

In this example, the query returns data from the table, **contacts**, that is in the database, **salesdb**.

A *remote* database server is any database server that is not the current database server. When the external database is on a remote database server, you must qualify the name of the database object with the database server name and the database name, as the following example illustrates:

```
SELECT name, number FROM salesdb@distantserver:contacts
```

In this example, the query returns data from the table, **contacts**, that is in the database, **salesdb** on the remote database server, **distantserver**.

For the syntax and rules on how to specify database object names in an external database, see the *HCL® Informix® Guide to SQL: Syntax*.

Access ANSI databases

In ANSI databases, the owner of the object is part of the object name: **ownername.objectname**. When both the current and external databases are ANSI databases, unless you are the owner of the object, you must include the owner name. The following SELECT statement shows a fully-qualified table name:

```
SELECT name, number FROM salesdb@aserver:ownername.contacts
```

Tip: You can always over-qualify an object name. That is, you can specify the full object name, database@servername:ownername.objectname, even in situations that do not require the full object name.

For more information about ANSI-compliant databases, refer to the *IBM® Informix® Database Design and Implementation Guide*.

Create joins between external database servers

You can use the same notation in a join. When you specify the database name explicitly, the long table names can become cumbersome unless you use aliases to shorten them, as the following example shows:

```
SELECT O.order_num, C.fname, C.lname
   FROM masterdb@central:customer C, sales@boston:orders O
   WHERE C.customer_num = O.Customer_num
```

Access external routines

To refer to a routine on a database server other than the current database server, qualify the routine name with the database server name and database name (and the owner name if the remote database is ANSI compliant), as the following SELECT statement illustrates:

```
SELECT name, salesdb@boston:how_long()
   FROM salesdb@boston:contacts
```

Restrictions for remote database access

This section summarizes the restrictions for remote database access.

SQL statements that access more than one database

Only the data manipulation language (DML) statements of SQL, and a subset of the data definition language (DDL) statements of SQL can reference database objects outside the local database from which the statement is issued, or in databases of server instances that are not the local HCL Informix® server instance.

You can run the following SQL statements across databases and across database server instances:

- CREATE DATABASE
- CREATE SYNONYM
- CREATE VIEW
- DATABASE

- DELETE
- DROP DATABASE
- EXECUTE FUNCTION
- EXECUTE PROCEDURE
- INFO
- INSERT
- LOAD
- LOCK TABLE
- MERGE
- SELECT
- UNLOAD
- UNLOCK TABLE
- UPDATE

Restriction:

To run each of these SQL statements successfully across databases or across database servers, the local database and the external databases must all have the same logging mode. For example, if the local database from which you issue a distributed query was created as MODE ANSI, any other database that the query accesses cannot be unlogged, and cannot use explicit transactions.

Return data types in cross-database operations

Distributed operations that use SQL statements or UDRs to access other databases of the local HCL Informix® database server instance can return values of these data types:

- Any built-in atomic data type that is not opaque
- The built-in opaque types BLOB, BOOLEAN, BSON, CLOB, JSON, and LVARCHAR
- DISTINCT types based on a non-opaque built-in atomic type, or on a built-in opaque type listed above
- User-defined data types (UDTs) that can be cast to built-in types.

The DISTINCT or UDT values above must all be explicitly cast to built-in data types, and all the DISTINCT types, UDTs, and casts must be defined identically in each of the participating databases.

These data types can be returned by SPL, C, and Java-language UDRs as parameters or as return values, if the UDRs are defined in all of the participating databases. The DISTINCT data types must have exactly the same data type hierarchy defined in all databases that participate in the distributed query.

A cross-database distributed query or other cross-database DML operation that accesses another database of the local Informix® database server will fail with an error if it references a table, view, or synonym that includes a column of any of the following data types:

- IMPEXP
- IMPEXPBIN
- LOLIST
- SENDRECV
- DISTINCT of any of the built-in opaque data types in this list
- Complex types, including COLLECTION, LIST, MULTISET, or SET, and named or unnamed ROW types.

This restriction against cross-database distributed operations that access tables with these built-in opaque or complex data types also applies to operations that access databases of two or more database server instances, which the next section describes.

Return data types in cross-server operations

A distributed query (or any other distributed DML operation or function call) across databases of two or more Informix® instances cannot return complex or large-object data types, nor most UDTs or opaque data types. Cross-server distributed queries, DML operations, and function calls can return only the following data types:

- Any non-opaque built-in data type
- BOOLEAN
- BSON
- JSON
- LVARCHAR
- DISTINCT of non-opaque built-in types
- DISTINCT of BOOLEAN or LVARCHAR
- DISTINCT of BSON or JSON
- DISTINCT of any of the DISTINCT types in this list.

The same cross-database DDL requirements, that all UDRs, casts, and DISTINCT data types have identical definitions in every participating database, also apply to distributed SQL operations across the databases of two or more Informix® database-server instances.

A cross-server DML operation that accesses a database of another Informix® instance will fail with an error. however, if it references a table object that includes a column of any of the following data types:

- BLOB
- BYTE
- CLIENTBINVAL
- CLOB
- IFX_LO_SPEC
- IFX_LO_STAT
- IMPEXP
- IMPEXPBIN
- INDEXKEYARRAY
- LOLIST

- POINTER
- RTNPARAMTYPES
- SELFUNCARGS
- SENDRECV
- STAT
- TEXT
- XID
- User-defined OPAQUE type
- Complex types, including COLLECTION, LIST, MULTISET, or SET, and named or unnamed ROW types.
- DISTINCT of any of the opaque or complex data types in this list.

Access external database objects

To access external database objects:

- You must hold appropriate access permissions on these objects.
- Both databases must be set to the same locale.

Important: Distributed transactions cannot access objects in a database of another Informix® server instance unless both server instances support either a TCP/IP or an IPCSTR connection, as defined in their DBSERVERNAME or DBSERVERALIASES configuration parameters and in the sqlhosts information. This connection-type requirement applies to any communication between Informix® database server instances, even if both database servers reside on the same computer.

SQL programming

The previous examples treat SQL as if it were an interactive computer language; that is, as if you could type a SELECT statement directly into the database server and see rows of data rolling back to you.

Of course, that is not the case. Many layers of software stand between you and the database server. The database server retains data in a binary form that must be formatted before it can be displayed. It does not return a mass of data at once; it returns one row at a time, as a program requests it.

You can access information in your database through interactive access with , through application programs written with an SQL API such as , or through an application language such as SPL.

Almost any program can contain SQL statements, execute them, and retrieve data from a database server. This chapter explains how these activities are performed and indicates how you can write programs that perform them.

This chapter introduces concepts that are common to SQL programming in any language. Before you can write a successful program in a particular programming language, you must first become fluent in that language. Then, because the details of the process are different in every language, you must become familiar with the publication for the Informix® SQL API specific to that language.

SQL in programs

You can write a program in any of several languages and mix SQL statements among the other statements of the program, just as if they were ordinary statements of that programming language. These SQL statements are embedded in the program, and the program contains *embedded SQL*, which is often abbreviated as .

SQL in SQL APIs

products are Informix® SQL APIs (application programming interfaces). IBM® produces an SQL API for the C programming language.

The following figure shows how an SQL API product works. You write a source program in which you treat SQL statements as executable code. Your source program is processed by an embedded SQL *preprocessor*, a program that locates the embedded SQL statements and converts them into a series of procedure calls and special data structures.
Figure 380. Overview of processing a program with embedded SQL statements



The converted source program then passes through the programming language compiler. The compiler output becomes an executable program after it is linked with a static or *dynamic* library of SQL API procedures. When the program runs, the SQL API library procedures are called; they set up communication with the database server to carry out the SQL operations.

If you link your executable program to a threading library package, you can develop *multithreaded applications*. A multithreaded application can have many threads of control. It separates a process into multiple execution threads, each of which runs independently. The major advantage of a multithreaded application is that each thread can have many active connections to a database server simultaneously. While a nonthreaded application can establish many connections to one or more databases, it can have only one connection active at a time. A multithreaded application can have one active connection per thread and many threads per application.

For more information on multithreaded applications, see the *HCL® Informix® Enterprise Replication Guide*.

SQL in application languages

Whereas Informix® SQL API products allow you to embed SQL in the host language, some languages include SQL as a natural part of their statement set. Informix® Stored Procedure Language (SPL) uses SQL as a natural part of its statement set. You use an SQL API product to write application programs. You use SPL to write routines that are stored with a database and called from an application program.

Static embedding

You can introduce SQL statements into a program through *static embedding* or *dynamic statements*. The simpler and more common way is by static embedding, which means that the SQL statements are written as part of the code. The statements

are *static* because they are a fixed part of the source text. For more information on static embedding, see Retrieve single rows on page 1707 and Retrieve multiple rows on page 1711.

Dynamic statements

Some applications require the ability to compose SQL statements *dynamically*, in response to user input. For example, a program might have to select different columns or apply different criteria to rows, depending on what the user wants.

With dynamic SQL, the program composes an SQL statement as a string of characters in memory and passes it to the database server to be executed. Dynamic statements are not part of the code; they are constructed in memory during execution. For more information, see Dynamic SQL on page 1717.

Program variables and host variables

Application programs can use program variables within SQL statements. In SPL, you put the program variable in the SQL statement as syntax allows. For example, a DELETE statement can use a program variable in its WHERE clause.

The following code example shows a program variable in SPL.

```
CREATE PROCEDURE delete_item (drop_number INT)

DELETE FROM items WHERE order_num = drop_number
```

In applications that use embedded SQL statements, the SQL statements can refer to the contents of program variables. A program variable that is named in an embedded SQL statement is called a *host variable* because the SQL statement is thought of as a guest in the program.

The following example shows a DELETE statement as it might appear when it is embedded in the source program:

```
EXEC SQL delete FROM items
   WHERE order_num = :onum;
```

In this program, you see an ordinary DELETE statement, as Modify data on page 1658 describes. When the program is executed, a row of the **items** table is deleted; multiple rows can also be deleted.

The statement contains one new feature. It compares the **order_num** column to an item written as **:onum**, which is the name of a host variable.

An SQL API product provides a way to delimit the names of host variables when they appear in the context of an SQL statement. In , a host variable can be introduced with either a dollar sign ($) or a colon (:). The colon is the ANSI-compatible format. The example statement asks the database server to delete rows in which the order number equals the current contents of the host variable named **:onum**. This numeric variable was declared and assigned a value earlier in the program.

In , an SQL statement can be introduced with either a leading dollar sign ($) or the words EXEC SQL.

The differences of syntax as illustrated in the preceding examples are trivial; the essential point is that the SQL API and SPL languages let you perform the following tasks:

- Embed SQL statements in a source program as if they were executable statements of the host language.
- Use program variables in SQL expressions the way literal values are used.

If you have programming experience, you can immediately see the possibilities. In the example, the order number to be deleted is passed in the variable **onum**. That value comes from any source that a program can use. It can be read from a file, the program can prompt a user to enter it, or it can be read from the database. The DELETE statement itself can be part of a subroutine (in which case **onum** can be a parameter of the subroutine); the subroutine can be called once or repetitively.

In short, when you embed SQL statements in a program, you can apply to them all the power of the host language. You can hide the SQL statements under many interfaces, and you can embellish the SQL functions in many ways.

Call the database server

Executing an SQL statement is essentially calling the database server as a subroutine. Information must pass from the program to the database server, and information must be returned from the database server to the program.

Some of this communication is done through host variables. You can think of the host variables named in an SQL statement as the parameters of the procedure call to the database server. In the preceding example, a host variable acts as a parameter of the WHERE clause. Host variables receive data that the database server returns, as Retrieve multiple rows on page 1711 describes.

SQL Communications Area

The database server always returns a result code, and possibly other information about the effect of an operation, in a data structure known as the SQL Communications Area (SQLCA). If the database server executes an SQL statement in a user-defined routine, the SQLCA of the calling application contains the values that the SQL statement triggers in the routine.

The principal fields of the SQLCA are listed in Table 86: Values of SQLCODE on page 1704 through Table 88: Fields of SQLWARN on page 1706. The syntax that you use to describe a data structure such as the SQLCA, as well as the syntax that you use to refer to a field in it, differs among programming languages. For details, see your SQL API publication.

In particular, the subscript by which you name one element of the SQLERRD and SQLWARN arrays differs. Array elements are numbered starting with zero in , but starting with one in other languages. In this discussion, the fields are named with specific words such as `third`, and you must translate these words into the syntax of your programming language.

You can also use the SQLSTATE variable of the GET DIAGNOSTICS statement to detect, handle, and diagnose errors. See SQLSTATE value on page 1707.

SQLCODE field

The SQLCODE field is the primary return code of the database server. After every SQL statement, SQLCODE is set to an integer value as the following table shows. When that value is zero, the statement is performed without error. In particular, when a statement is supposed to return data into a host variable, a code of zero means that the data has been returned and can be used. Any nonzero code means the opposite. No useful data was returned to host variables.

Table 86. Values of SQLCODE

Return value	Interpretation
value < 0	Specifies an error code.
value = 0	Indicates success.
0 < *value* < 100	After a DESCRIBE statement, an integer value that represents the type of SQL statement that is described.
100	After a successful query that returns no rows, indicates the NOT FOUND condition. NOT FOUND can also occur in an ANSI-compliant database after an INSERT INTO/SELECT, UPDATE, DELETE, or SELECT... INTO TEMP statement fails to access any rows.

End of data

The database server sets SQLCODE to `100` when the statement is performed correctly but no rows are found. This condition can occur in two situations.

The first situation involves a query that uses a cursor. (Retrieve multiple rows on page 1711 describes queries that use cursors.) In these queries, the FETCH statement retrieves each value from the active set into memory. After the last row is retrieved, a subsequent FETCH statement cannot return any data. When this condition occurs, the database server sets SQLCODE to `100`, which indicates `end of data, no rows found`.

The second situation involves a query that does not use a cursor. In this case, the database server sets SQLCODE to `100` when no rows satisfy the query condition. In databases that are not ANSI compliant, only a SELECT statement that returns no rows causes SQLCODE to be set to `100`.

In ANSI-compliant databases, SELECT, DELETE, UPDATE, and INSERT statements all set SQLCODE to `100` if no rows are returned.

Negative Codes

When something unexpected goes wrong during a statement, the database server returns a negative number in SQLCODE to explain the problem. The meanings of these codes are documented in the online error message file.

SQLERRD array

Some error codes that can be reported in SQLCODE reflect general problems. The database server can set a more detailed code in the second field of SQLERRD that reveals the error that the database server I/O routines or the operating system encountered.

The bigints in the SQLERRD array are set to different values following different statements. The first and fourth elements of the array are used only in . The following table shows how the fields are used.

Table 87. Fields of SQLERRD

Field	Interpretation
First	After a successful PREPARE statement for a SELECT, UPDATE, INSERT, or DELETE statement, or after a Select cursor is opened, this field contains the estimated number of rows affected.
Second	When SQLCODE contains an error code, this field contains either zero or an additional error code, called the error code, that explains the cause of the main error. After a successful insert operation of a single row, this field contains the value of any SERIAL, BIGSERIAL, or SERIAL8 value generated for that row. (This field is not updated, however, when a serial column is directly inserted as a triggered action by a trigger on a table, or by an INSTEAD OF trigger on a view.)
Third	After a successful multirow insert, update, or delete operation, this field contains the number of rows that were processed. After a multirow insert, update, or delete operation that ends with an error, this field contains the number of rows that were successfully processed before the error was detected.
Fourth	After a successful PREPARE statement for a SELECT, UPDATE, INSERT, or DELETE statement, or after a select cursor has been opened, this field contains the estimated weighted sum of disk accesses and total rows processed.
Fifth	After a syntax error in a PREPARE, EXECUTE IMMEDIATE, DECLARE, or static SQL statement, this field contains the offset in the statement text where the error was detected.
Sixth	After a successful fetch of a selected row, or a successful insert, update, or delete operation, this field contains the rowid (physical address) of the last row that was processed. Whether this rowid value corresponds to a row that the database server returns to the user depends on how the database server processes a query, particularly for SELECT statements.
Seventh	Reserved.

These additional details can be useful. For example, you can use the value in the third field to report how many rows were deleted or updated. When your program prepares an SQL statement that the user enters and an error is found, the value in the fifth field enables you to display the exact point of error to the user. (uses this feature to position the cursor when you ask to modify a statement after an error.)

SQLWARN array

The eight character fields in the SQLWARN array are set to either a blank or to `w` to indicate a variety of special conditions. Their meanings depend on the statement just executed.

A set of warning flags appears when a database opens, that is, following a CONNECT, DATABASE, or CREATE DATABASE statement. These flags tell you some characteristics of the database as a whole.

A second set of flags appears following any other statement. These flags reflect unusual events that occur during the statement, which are usually not serious enough to be reflected by SQLCODE.

Both sets of SQLWARN values are summarized in the following table.

Table 88. Fields of SQLWARN

Field	When opening or connecting to a database	All other SQL operations
First	Set to `w` when any other warning field is set to `w`. If blank, others need not be checked.	Set to `w` when any other warning field is set to `w`.
Second	Set to `w` when the database now open uses a transaction log.	Set to `w` if a column value is truncated when it is fetched into a host variable using a FETCH or a SELECT...INTO statement. On a REVOKE ALL statement, set to `w` when not all seven table-level privileges are revoked.
Third	Set to `w` when the database now open is ANSI compliant.	Set to `w` when a FETCH or SELECT statement returns an aggregate function (SUM, AVG, MIN, MAX) value that is NULL.
Fourth	Set to `w` when the database server is HCL Informix®.	On a SELECT ... INTO, FETCH ... INTO, or EXECUTE ... INTO statement, set to `w` when the number of projection list items is not the same as the number of host variables given in the INTO clause to receive them. On a GRANT ALL statement, set to `w` when not all seven table-level access privileges are granted.
Fifth	Set to `w` when the database server stores the FLOAT data type in DECIMAL form (done when the host system lacks support for FLOAT types).	Set to `w` after a DESCRIBE statement if the prepared object contains a DELETE statement or an UPDATE statement without a WHERE clause.
Sixth	Reserved.	Set to `w` following execution of a statement that does not use ANSI-standard SQL syntax (provided the **DBANSIWARN** environment variable is set).
Seventh	Set to `w` when the application is connected to a database server that is the secondary server in a data-replication pair. That is, the server is available only for read operations.	Set to `w` when a data fragment (a dbspace) has been skipped during query processing (when the DATASKIP feature is on).
Eighth	Set to `w` when client DB_LOCALE does not match the database locale. For more information, see the *HCL® Informix® GLS User's Guide*.	Set to `w` when SET EXPLAIN ON AVOID_EXECUTE statement prevents query execution.

SQLERRM character string

SQLERRM can store a character string of up to 72 bytes. The SQLERRM character string contains identifiers, such as a table names, that are placed in the error message. For some networked applications, it contains an error message that the networking software generates.

If an INSERT operation fails because a constraint is violated, the name of the constraint that failed is written to SQLERRM.

Tip: If an error string is longer than 72 bytes, the overflow is silently discarded. In some contexts, this can result in the loss of information about runtime errors.

SQLSTATE value

Certain HCL® Informix® products, such as , support the SQLSTATE value in compliance with X/Open and ANSI SQL standards. The GET DIAGNOSTICS statement reads the SQLSTATE value to diagnose errors after you run an SQL statement. The database server returns a result code in a five-character string that is stored in a variable called SQLSTATE. The SQLSTATE error code, or value, tells you the following information about the most recently executed SQL statement:

- If the statement was successful
- If the statement was successful but generated warnings
- If the statement was successful but generated no data
- If the statement failed

For more information on the GET DIAGNOSTICS statement, the SQLSTATE variable, and the meaning of the SQLSTATE return codes, see the GET DIAGNOSTICS statement in the *HCL® Informix® Guide to SQL: Syntax*.

Tip: If your HCL® Informix® product supports GET DIAGNOSTICS and SQLSTATE, it is recommended that you use them as the primary structure to detect, handle, and diagnose errors. Using SQLSTATE allows you to detect multiple errors, and it is ANSI compliant.

Retrieve single rows

The set of rows that a SELECT statement returns is its *active set*. A *singleton* SELECT statement returns a single row. You can use embedded SELECT statements to retrieve single rows from the database into host variables. When a SELECT statement returns more than one row of data, however, a program must use a *cursor* to retrieve rows one at a time. Multiple-row select operations are discussed in Retrieve multiple rows on page 1711.

To retrieve a single row of data, simply embed a SELECT statement in your program. The following example shows how you can write the embedded SELECT statement using :

```
EXEC SQL SELECT avg (total_price)
   INTO :avg_price
   FROM items
   WHERE order_num in
      (SELECT order_num from orders
      WHERE order_date < date('6/1/98') );
```

The INTO clause is the only detail that distinguishes this statement from any example in Compose SELECT statements on page 1532 or Compose advanced SELECT statements on page 1619. This clause specifies the host variables that are to receive the data that is produced.

When the program executes an embedded SELECT statement, the database server performs the query. The example statement selects an aggregate value so that it produces exactly one row of data. The row has only a single column, and its value is deposited in the host variable named **avg_price**. Subsequent lines of the program can use that variable.

You can use statements of this kind to retrieve single rows of data into host variables. The single row can have as many columns as desired. If a query produces more than one row of data, the database server cannot return any data. It returns an error code instead.

You should list as many host variables in the INTO clause as there are items in the select list. If, by accident, these lists are of different lengths, the database server returns as many values as it can and sets the warning flag in the fourth field of SQLWARN.

Data type conversion

The following example retrieves the average of a DECIMAL column, which is itself a DECIMAL value. However, the host variable into which the average of the DECIMAL column is placed is not required to have that data type.

```
EXEC SQL SELECT avg (total_price) into :avg_price
   FROM items;
```

The declaration of the receiving variable **avg_price** in the previous example of code is not shown. The declaration could be any one of the following definitions:

```
int avg_price;
double avg_price;
char avg_price[16];
dec_t avg_price; /* typedef of decimal number structure */
```

The data type of each host variable that is used in a statement is noted and passed to the database server with the statement. The database server does its best to convert column data into the form that the receiving variables use. Almost any conversion is allowed, although some conversions cause a precision loss. The results of the preceding example differ, depending on the data type of the receiving host variable, as the following table shows.

Data type	Result
FLOAT	The database server converts the decimal result to FLOAT, possibly truncating some fractional digits. If the magnitude of a decimal exceeds the maximum magnitude of the FLOAT format, an error is returned.
INTEGER	The database server converts the result to INTEGER, truncating fractional digits if necessary. If the integer part of the converted number does not fit the receiving variable, an error occurs.
CHARACTER	The database server converts the decimal value to a CHARACTER string. If the string is too long for the receiving variable, it is truncated. The second field of SQLWARN is set to W and the value in the SQLSTATE variable is 01004.

What if the program retrieves a NULL value?

NULL values can be stored in the database, but the data types that programming languages support do not recognize a NULL state. A program must have some way to recognize a NULL item to avoid processing it as data.

Indicator variables meet this need in SQL APIs. An indicator variable is an additional variable that is associated with a host variable that might receive a NULL item. When the database server puts data in the main variable, it also puts a special value in the indicator variable to show whether the data is NULL. In the following example, a single row is selected, and a single value is retrieved into the host variable **op_date**:

```
EXEC SQL SELECT paid_date
      INTO :op_date:op_d_ind
      FROM orders
      WHERE order_num = $the_order;
if (op_d_ind < 0) /* data was null */
   rstrdate ('01/01/1900', :op_date);
```

Because the value might be NULL, an indicator variable named **op_d_ind** is associated with the host variable. (It must be declared as a short integer elsewhere in the program.)

Following execution of the SELECT statement, the program tests the indicator variable for a negative value. A negative number (usually `-1`) means that the value retrieved into the main variable is NULL. If the variable is NULL, this program uses the library function to assign a default value to the host variable. (The function rstrdate is part of the product.)

The syntax that you use to associate an indicator variable with a host variable differs with the language you are using, but the principle is the same in all languages.

Dealing with errors

Although the database server automatically handles conversion between data types, several things still can go wrong with a SELECT statement. In SQL programming, as in any kind of programming, you must anticipate errors and provide for them at every point.

End of data

One common event is that no rows satisfy a query. This event is signalled by an SQLSTATE code of `02000` and by a code of `100` in SQLCODE after a SELECT statement. This code indicates an error or a normal event, depending entirely on your application. If you are sure a row or rows should satisfy the query (for example, if you are reading a row using a key value that you just read from a row of another table), then the end-of-data code represents a serious failure in the logic of the program. On the other hand, if you select a row based on a key that a user supplies or some other source supplies that is less reliable than a program, a lack of data can be a normal event.

End of data with databases that are not ANSI compliant

If your database is not ANSI compliant, the end-of-data return code, `100`, is set in SQLCODE following SELECT statements only. In addition, the SQLSTATE value is set to `02000`. (Other statements, such as INSERT, UPDATE, and DELETE, set the third element of SQLERRD to show how many rows they affected; Modify data through SQL programs on page 1723 covers this topic.)

Serious errors

Errors that set SQLCODE to a negative value or SQLSTATE to a value that begins with anything other than `00`, `01`, or `02` are usually serious. Programs that you have developed and that are in production should rarely report these errors. Nevertheless, it is difficult to anticipate every problematic situation, so your program must be able to deal with these errors.

For example, a query can return error -206, which means that a table specified in the query is not in the database. This condition occurs if someone dropped the table after the program was written, or if the program opened the wrong database through some error of logic or mistake in input.

Interpret end of data with aggregate functions

A SELECT statement that uses an aggregate function such as SUM, MIN, or AVG always succeeds in returning at least one row of data, even when no rows satisfy the WHERE clause. An aggregate value based on an empty set of rows is null, but it exists nonetheless.

However, an aggregate value is also null if it is based on one or more rows that all contain null values. If you must be able to detect the difference between an aggregate value that is based on no rows and one that is based on some rows that are all null, you must include a COUNT function in the statement and an indicator variable on the aggregate value. You can then work out the following cases.

Count Value	Indicator	Case
0	-1	Zero rows selected
>0	-1	Some rows selected; all were null
>0	0	Some non-null rows selected

Default values

You can handle these inevitable errors in many ways. In some applications, more lines of code are used to handle errors than to execute functionality. In the examples in this section, however, one of the simplest solutions, the default value, should work, as the following example shows:

```
avg_price = 0; /* set default for errors */
EXEC SQL SELECT avg (total_price)
      INTO :avg_price:null_flag
      FROM items;
if (null_flag < 0) /* probably no rows */
   avg_price = 0; /* set default for 0 rows */
```

The previous example deals with the following considerations:

- If the query selects some non-null rows, the correct value is returned and used. This result is the expected and most frequent one.
- If the query selects no rows, or in the much less likely event, selects only rows that have null values in the **total_price** column (a column that should never be null), the indicator variable is set, and the default value is assigned.
- If any serious error occurs, the host variable is left unchanged; it contains the default value initially set. At this point in the program, the programmer sees no need to trap such errors and report them.

Retrieve multiple rows

When any chance exists that a query could return more than one row, the program must execute the query differently. Multirow queries are handled in two stages. First, the program starts the query. (No data is returned immediately.) Then the program requests the rows of data one at a time.

These operations are performed using a special data object called a *cursor*. A cursor is a data structure that represents the current state of a query. The following list shows the general sequence of program operations:

1. The program *declares* the cursor and its associated SELECT statement, which merely allocates storage to hold the cursor.
2. The program *opens* the cursor, which starts the execution of the associated SELECT statement and detects any errors in it.
3. The program *fetches* a row of data into host variables and processes it.
4. The program *closes* the cursor after the last row is fetched.
5. When the cursor is no longer needed, the program *frees* the cursor to deallocate the resources it uses.

These operations are performed with SQL statements named DECLARE, OPEN, FETCH, CLOSE, and FREE.

Declare a cursor

You use the DECLARE statement to declare a cursor. This statement gives the cursor a name, specifies its use, and associates it with a statement. The following example is written in :

```
EXEC SQL DECLARE the_item CURSOR FOR
   SELECT order_num, item_num, stock_num
      INTO :o_num, :i_num, :s_num
      FROM items
   FOR READ ONLY;
```

The declaration gives the cursor a name (**the_item** in this case) and associates it with a SELECT statement. (Modify data through SQL programs on page 1723 discusses how a cursor can also be associated with an INSERT statement.)

The SELECT statement in this example contains an INTO clause. The INTO clause specifies which variables receive data. You can also use the FETCH statement to specify which variables receive data, as Locate the INTO clause on page 1713 discusses.

The DECLARE statement is not an active statement; it merely establishes the features of the cursor and allocates storage for it. You can use the cursor declared in the preceding example to read through the **items** table once. Cursors can be declared to read backward and forward (see Cursor input modes on page 1713). This cursor, because it lacks a FOR UPDATE clause

and because it is designated FOR READ ONLY, is used only to read data, not to modify it. Modify data through SQL programs on page 1723 covers the use of cursors to modify data.

Open a cursor

The program opens the cursor when it is ready to use it. The OPEN statement activates the cursor. It passes the associated SELECT statement to the database server, which begins the search for matching rows. The database server processes the query to the point of locating or constructing the first row of output. It does not actually return that row of data, but it does set a return code in SQLSTATE and in SQLCODE for SQL APIs. The following example shows the OPEN statement in :

```
EXEC SQL OPEN the_item;
```

Because the database server is seeing the query for the first time, it might detect a number of errors. After the program opens the cursor, it should test SQLSTATE or SQLCODE. If the SQLSTATE value is greater than `02000` or the SQLCODE contains a negative number, the cursor is not usable. An error might be present in the SELECT statement, or some other problem might prevent the database server from executing the statement.

If SQLSTATE is equal to `00000`, or SQLCODE contains a zero, the SELECT statement is syntactically valid, and the cursor is ready to use. At this point, however, the program does not know if the cursor can produce any rows.

Fetch rows

The program uses the FETCH statement to retrieve each row of output. This statement names a cursor and can also name the host variables that receive the data. The following example shows the completed code:

```
EXEC SQL DECLARE the_item CURSOR FOR
   SELECT order_num, item_num, stock_num
      INTO :o_num, :i_num, :s_num
      FROM items;
EXEC SQL OPEN the_item;
while(SQLCODE == 0)
{
   EXEC SQL FETCH the_item;
   if(SQLCODE == 0)
      printf("%d, %d, %d", o_num, i_num, s_num);
}
```

Detect end of data

In the previous example, the WHILE condition prevents execution of the loop in case the OPEN statement returns an error. The same condition terminates the loop when SQLCODE is set to `100` to signal the end of data. However, the loop contains a test of SQLCODE. This test is necessary because, if the SELECT statement is valid yet finds no matching rows, the OPEN statement returns a zero, but the first fetch returns `100` (end of data) and no data. The following example shows another way to write the same loop:

```
EXEC SQL DECLARE the_item CURSOR FOR
   SELECT order_num, item_num, stock_num
   INTO :o_num, :i_num, :s_num
   FROM items;
EXEC SQL OPEN the_item;
if(SQLCODE == 0)
```

```
   EXEC SQL FETCH the_item;      /* fetch 1st row*/
while(SQLCODE == 0)
{
   printf("%d, %d, %d", o_num, i_num, s_num);
   EXEC SQL FETCH the_item;
}
```

In this version, the case of no returned rows is handled early, so no second test of SQLCODE exists within the loop. These versions have no measurable difference in performance because the time cost of a test of SQLCODE is a tiny fraction of the cost of a fetch.

Locate the INTO clause

The INTO clause names the host variables that are to receive the data that the database server returns. The INTO clause must appear in either the SELECT or the FETCH statement. However it cannot appear in both statements. The following example specifies host variables in the FETCH statement:

```
EXEC SQL DECLARE the_item CURSOR FOR
   SELECT order_num, item_num, stock_num
      FROM items;
EXEC SQL OPEN the_item;
while(SQLCODE == 0)
{
   EXEC SQL FETCH the_item INTO :o_num, :i_num, :s_num;
   if(SQLCODE == 0)
      printf("%d, %d, %d", o_num, i_num, s_num);
}
```

This form lets you fetch different rows into different locations. For example, you could use this form to fetch successive rows into successive elements of an array.

Cursor input modes

For purposes of input, a cursor operates in one of two modes, *sequential* or *scrolling*. A sequential cursor can fetch only the next row in sequence, so a sequential cursor can read through a table only once each time the cursor is opened. A scroll cursor can fetch the next row or any of the output rows, so a scroll cursor can read the same rows multiple times. The following example shows a sequential cursor declared in .

```
EXEC SQL DECLARE pcurs cursor for
   SELECT customer_num, lname, city
      FROM customer;
```

After the cursor is opened, it can be used only with a sequential fetch that retrieves the next row of data, as the following example shows:

```
EXEC SQL FETCH p_curs into:cnum, :clname, :ccity;
```

Each sequential fetch returns a new row.

A scroll cursor is declared with the keywords SCROLL CURSOR, as the following example from shows:

```
EXEC SQL DECLARE s_curs SCROLL CURSOR FOR
      SELECT order_num, order_date FROM orders
         WHERE customer_num > 104
```

Use the scroll cursor with a variety of fetch options. For example, the ABSOLUTE option specifies the absolute row position of the row to fetch.

```
EXEC SQL FETCH ABSOLUTE :numrow s_curs
     INTO :nordr, :nodat
```

This statement fetches the row whose position is given in the host variable **numrow**. You can also fetch the current row again, or you can fetch the first row and then scan through all the rows again. However, these features can cause the application to run more slowly, as the next section describes. For additional options that apply to scroll cursors, see the FETCH statement in the *HCL® Informix® Guide to SQL: Syntax*.

Active set of a cursor

Once a cursor is opened, it stands for some selection of rows. The set of all rows that the query produces is called the *active set* of the cursor. It is easy to think of the active set as a well-defined collection of rows and to think of the cursor as pointing to one row of the collection. This situation is true as long as no other programs are modifying the same data concurrently.

Create the active set

When a cursor is opened, the database server does whatever is necessary to locate the first row of selected data. Depending on how the query is phrased, this action can be easy, or it can require a great deal of work and time. Consider the following declaration of a cursor:

```
EXEC SQL DECLARE easy CURSOR FOR
   SELECT fname, lname FROM customer
      WHERE state = 'NJ'
```

Because this cursor queries only a single table in a simple way, the database server quickly determines whether any rows satisfy the query and identifies the first one. The first row is the only row the cursor finds at this time. The rest of the rows in the active set remain unknown. As a contrast, consider the following declaration of a cursor:

```
EXEC SQL DECLARE hard SCROLL CURSOR FOR
   SELECT C.customer_num, O.order_num, sum (items.total_price)
      FROM customer C, orders O, items I
      WHERE C.customer_num = O.customer_num
         AND O.order_num = I.order_num
         AND O.paid_date is null
      GROUP BY C.customer_num, O.order_num
```

The active set of this cursor is generated by joining three tables and grouping the output rows. The optimizer might be able to use indexes to produce the rows in the correct order, but generally the use of ORDER BY or GROUP BY clauses requires the database server to generate all the rows, copy them to a temporary table, and sort the table, before it can determine which row to present first.

In cases where the active set is entirely generated and saved in a temporary table, the database server can take quite some time to open the cursor. Afterwards, the database server could tell the program exactly how many rows the active

set contains. However, this information is not made available. One reason is that you can never be sure which method the optimizer uses. If the optimizer can avoid sorts and temporary tables, it does so; but small changes in the query, in the sizes of the tables, or in the available indexes can change the methods of the optimizer.

Active set for a sequential cursor

The database server attempts to use as few resources as possible to maintain the active set of a cursor. If it can do so, the database server never retains more than the single row that is fetched next. It can do this for most sequential cursors. On each fetch, it returns the contents of the current row and locates the next one.

Active set for a SCROLL cursor

All the rows in the active set for a SCROLL cursor must be retained until the cursor closes because the database server cannot be sure which row the program will ask for next.

Most frequently, the database server implements the active set of a scroll cursor as a temporary table. The database server might not fill this table immediately, however (unless it created a temporary table to process the query). Usually it creates the temporary table when the cursor is opened. Then, the first time a row is fetched, the database server copies it into the temporary table and returns it to the program. When a row is fetched for a second time, it can be taken from the temporary table. This scheme uses the fewest resources, in the event that the program abandons the query before it fetches all the rows. Rows that are never fetched are not created or saved.

Active set and concurrency

When only one program is using a database, the members of the active set cannot change. This situation describes most personal computers, and it is the easiest situation to think about. But some programs must be designed for use in a multiprogramming system, where two, three, or dozens of different programs can work on the same tables simultaneously.

When other programs can update the tables while your cursor is open, the idea of the active set becomes less useful. Your program can see only one row of data at a time, but all other rows in the table can be changing.

In the case of a simple query, when the database server holds only one row of the active set, any other row can change. The instant after your program fetches a row, another program can delete the same row, or update it so that if it is examined again, it is no longer part of the active set.

When the active set, or part of it, is saved in a temporary table, *stale data* can present a problem. That is, the rows in the actual tables from which the active-set rows are derived can change. If they do, some of the active-set rows no longer reflect the current table contents.

These ideas seem unsettling at first, but as long as your program only reads the data, stale data does not exist, or rather, all data is equally stale. The active set is a snapshot of the data as it is at one moment. A row is different the next day; it does not matter if it is also different in the next millisecond. To put it another way, no practical difference exists between changes that occur while the program is running and changes that are saved and applied the instant that the program terminates.

The only time that stale data can cause a problem is when the program intends to use the input data to modify the same database; for example, when a banking application must read an account balance, change it, and write it back. Modify data through SQL programs on page 1723 discusses programs that modify data.

Parts-explosion problem

When you use a cursor supplemented by program logic, you can solve problems that plain SQL cannot solve. One of these problems is the parts-explosion problem, sometimes called bill-of-materials processing. At the heart of this problem is a recursive relationship among objects; one object contains other objects, which contain yet others.

The problem is usually stated in terms of a manufacturing inventory. A company makes a variety of parts, for example. Some parts are discrete, but some are assemblages of other parts.

These relationships are documented in a single table, which might be called **contains**. The column **contains.parent** holds the part numbers of parts that are assemblages. The column **contains.child** has the part number of a part that is a component of the parent. If part number 123400 is an assembly of nine parts, nine rows exist with 123400 in the first column and other part numbers in the second. The following figure shows one of the rows that describe part number 123400.

Figure 381. Parts-explosion problem

CONTAINS

PARENT	CHILD
FKNN	FKNN
123400	432100
432100	765899

Here is the parts-explosion problem: given a part number, produce a list of all parts that are components of that part. The following example is a sketch of one solution, as implemented in :

```
int part_list[200];

boom(top_part)
int top_part;
{
   long this_part, child_part;
   int next_to_do = 0, next_free = 1;
   part_list[next_to_do] = top_part;

   EXEC SQL DECLARE part_scan CURSOR FOR
      SELECT child INTO child_part FROM contains
         WHERE parent = this_part;
   while(next_to_do < next_free)
   {
      this_part = part_list[next_to_do];
      EXEC SQL OPEN part_scan;
      while(SQLCODE == 0)
      {
         EXEC SQL FETCH part_scan;
         if(SQLCODE == 0)
         {
```

```
            part_list[next_free] = child_part;
            next_free += 1;
         }
      }
      EXEC SQL CLOSE part_scan;
      next_to_do += 1;
   }
   return (next_free - 1);
}
```

Technically speaking, each row of the **contains** table is the head node of a directed acyclic graph, or *tree*. The function performs a breadth-first search of the tree whose root is the part number passed as its parameter. The function uses a cursor named **part_scan** to return all the rows with a particular value in the **parent** column. The innermost `while` loop opens the **part_scan** cursor, fetches each row in the selection set, and closes the cursor when the part number of each component has been retrieved.

This function addresses the heart of the parts-explosion problem, but the function is not a complete solution. For example, it does not allow for components that appear at more than one level in the tree. Furthermore, a practical **contains** table would also have a column **count**, giving the count of **child** parts used in each **parent**. A program that returns a total count of each component part is much more complicated.

The iterative approach described previously is not the only way to approach the parts-explosion problem. If the number of generations has a fixed limit, you can solve the problem with a single SELECT statement using nested, outer self-joins.

If up to four generations of parts can be contained within one top-level part, the following SELECT statement returns all of them:

```
SELECT a.parent, a.child, b.child, c.child, d.child
   FROM contains a
      OUTER (contains b,
         OUTER (contains c, outer contains d) )
   WHERE a.parent = top_part_number
      AND a.child = b.parent
      AND b.child = c.parent
      AND c.child = d.parent
```

This SELECT statement returns one row for each line of descent rooted in the part given as **top_part_number**. Null values are returned for levels that do not exist. (Use indicator variables to detect them.) To extend this solution to more levels, select additional nested outer joins of the **contains** table. You can also revise this solution to return counts of the number of parts at each level.

Dynamic SQL

Although static SQL is useful, it requires that you know the exact content of every SQL statement at the time you write the program. For example, you must state exactly which columns are tested in any WHERE clause and exactly which columns are named in any select list.

No problem exists when you write a program to perform a well-defined task. But the database tasks of some programs cannot be perfectly defined in advance. In particular, a program that must respond to an interactive user might need to compose SQL statements in response to what the user enters.

Dynamic SQL allows a program to form an SQL statement during execution, so that user input determines the contents of the statement. This action is performed in the following steps:

1. The program assembles the text of an SQL statement as a character string, which is stored in a program variable.
2. It executes a PREPARE statement, which asks the database server to examine the statement text and prepare it for execution.
3. It uses the EXECUTE statement to execute the prepared statement.

In this way, a program can construct and then use any SQL statement, based on user input of any kind. For example, it can read a file of SQL statements and prepare and execute each one.

, a utility that you can use to explore SQL interactively, is the program that constructs, prepares, and executes SQL statements dynamically. For example, lets you use simple, interactive menus to specify the columns of a table. When you are finished, builds the necessary CREATE TABLE or ALTER TABLE statement dynamically and prepares and executes it.

Prepare a statement

In form, a dynamic SQL statement is like any other SQL statement that is written into a program, except that it cannot contain the names of any host variables.

A prepared SQL statement has two restrictions. First, if it is a SELECT statement, it cannot include the INTO *variable* clause. The INTO *variable* clause specifies host variables into which column data is placed, and host variables are not allowed in the text of a prepared object. Second, wherever the name of a host variable normally appears in an expression, a question mark (?) is written as a placeholder in the PREPARE statement. Only the PREPARE statement can specify question mark (?) placeholders.

You can prepare a statement in this form for execution with the PREPARE statement. The following example is written in :

```
EXEC SQL prepare query_2 from
       'SELECT * from orders
          WHERE customer_num = ? and order_date > ?';
```

The two question marks in this example indicate that when the statement is executed, the values of host variables are used at those two points.

You can prepare almost any SQL statement dynamically. The only statements that you cannot prepare are the ones directly concerned with dynamic SQL and cursor management, such as the PREPARE and OPEN statements. After you prepare an UPDATE or DELETE statement, it is a good idea to test the fifth field of SQLWARN to see if you used a WHERE clause (see SQLWARN array on page 1705).

The result of preparing a statement is a data structure that represents the statement. This data structure is not the same as the string of characters that produced it. In the PREPARE statement, you give a name to the data structure; it is **query_2** in the preceding example. This name is used to execute the prepared SQL statement.

The PREPARE statement does not limit the character string to one statement. It can contain multiple SQL statements, separated by semicolons. The following example shows a fairly complex transaction in :

```
strcpy(big_query, "UPDATE account SET balance = balance + ?
WHERE customer_id = ?; \ UPDATE teller SET balance =
balance + ? WHERE teller_id = ?;");
EXEC SQL PREPARE big1 FROM :big_query;
```

When this list of statements is executed, host variables must provide values for six place-holding question marks. Although it is more complicated to set up a multistatement list, performance is often better because fewer exchanges take place between the program and the database server.

Execute prepared SQL

After you prepare a statement, you can execute it multiple times. statements other than SELECT statements, and SELECT statements that return only a single row, are executed with the EXECUTE statement.

The following code prepares and executes a multistatement update of a bank account:

```
EXEC SQL BEGIN DECLARE SECTION;
char bigquery[270] = "begin work;";
EXEC SQL END DECLARE SECTION;
stcat ("update account set balance = balance + ? where ", bigquery);
stcat ("acct_number = ?;', bigquery);
stcat ("update teller set balance = balance + ? where ", bigquery);
stcat ("teller_number = ?;', bigquery);
stcat ("update branch set balance = balance + ? where ", bigquery);
stcat ("branch_number = ?;', bigquery);
stcat ("insert into history values(timestamp, values);", bigquery);

EXEC SQL prepare bigq from :bigquery;

EXEC SQL execute bigq using :delta, :acct_number, :delta,
    :teller_number, :delta, :branch_number;

EXEC SQL commit work;
```

The USING clause of the EXECUTE statement supplies a list of host variables whose values are to take the place of the question marks in the prepared statement. If a SELECT (or EXECUTE FUNCTION) returns only one row, you can use the INTO clause of EXECUTE to specify the host variables that receive the values.

Dynamic host variables

SQL APIs, which support dynamically allocated data objects, take dynamic statements one step further. They let you dynamically allocate the host variables that receive column data.

Dynamic allocation of variables makes it possible to take an arbitrary SELECT statement from program input, determine how many values it produces and their data types, and allocate the host variables of the appropriate types to hold them.

The key to this ability is the DESCRIBE statement. It takes the name of a prepared SQL statement and returns information about the statement and its contents. It sets SQLCODE to specify the type of statement; that is, the verb with which it begins. If the prepared statement is a SELECT statement, the DESCRIBE statement also returns information about the selected output data. If the prepared statement is an INSERT statement, the DESCRIBE statement returns information about the input parameters. The data structure to which a DESCRIBE statement returns information is a predefined data structure that is

allocated for this purpose and is known as a system-descriptor area. If you are using , you can use a system-descriptor area or, as an alternative, an **sqlda** structure.

The data structure that a DESCRIBE statement returns or references for a SELECT statement includes an array of structures. Each structure describes the data that is returned for one item in the select list. The program can examine the array and discover that a row of data includes a decimal value, a character value of a certain length, and an integer.

With this information, the program can allocate memory to hold the retrieved values and put the necessary pointers in the data structure for the database server to use.

Free prepared statements

A prepared SQL statement occupies space in memory. With some database servers, it can consume space that the database server owns as well as space that belongs to the program. This space is released when the program terminates, but in general, you should free this space when you finish with it.

You can use the FREE statement to release this space. The FREE statement takes either the name of a statement or the name of a cursor that was declared for a statement name, and releases the space allocated to the prepared statement. If more than one cursor is defined on the statement, freeing the statement does not free the cursor.

Quick execution

For simple statements that do not require a cursor or host variables, you can combine the actions of the PREPARE, EXECUTE, and FREE statements into a single operation. The following example shows how the EXECUTE IMMEDIATE statement takes a character string, prepares it, executes it, and frees the storage in one operation:

```
EXEC SQL execute immediate 'drop index my_temp_index';
```

This capability makes it easy to write simple SQL operations. However, because no USING clause is allowed, the EXECUTE IMMEDIATE statement cannot be used for SELECT statements.

Embed data-definition statements

Data-definition statements, the SQL statements that create databases and modify the definitions of tables, are not usually put into programs. The reason is that they are rarely performed. A database is created once, but it is queried and updated many times.

The creation of a database and its tables is generally done interactively, using . These tools can also be run from a file of statements, so that the creation of a database can be done with one operating-system command. The data-definition statements are documented in the *HCL® Informix® Guide to SQL: Syntax* and the *IBM® Informix® Database Design and Implementation Guide*.

Grant and revoke privileges in applications

One task related to data definition is performed repeatedly: granting and revoking privileges. Because privileges must be granted and revoked frequently, possibly by users who are not skilled in SQL, one strategy is to package the GRANT and REVOKE statements in programs to give them a simpler, more convenient user interface.

The GRANT and REVOKE statements are especially good candidates for dynamic SQL. Each statement takes the following parameters:

- A list of one or more privileges
- A table name
- The name of a user

You probably need to supply at least some of these values based on program input (from the user, command-line parameters, or a file) but none can be supplied in the form of a host variable. The syntax of these statements does not allow host variables at any point.

An alternative is to assemble the parts of a statement into a character string and to prepare and execute the assembled statement. Program input can be incorporated into the prepared statement as characters.

The following function assembles a GRANT statement from parameters, and then prepares and executes it:

```
char priv_to_grant[100];
char table_name[20];
char user_id[20];

table_grant(priv_to_grant, table_name, user_id)
char *priv_to_grant;
char *table_name;
char *user_id;
{
   EXEC SQL BEGIN DECLARE SECTION;
   char grant_stmt[200];
   EXEC SQL END DECLARE SECTION;

   sprintf(grant_stmt, " GRANT %s ON %s TO %s",
      priv_to_grant, table_name, user_id);
   PREPARE the_grant FROM :grant_stmt;
   if(SQLCODE == 0)
      EXEC SQL EXECUTE the_grant;
   else
      printf("Sorry, got error # %d attempting %s",
         SQLCODE, grant_stmt);

   EXEC SQL FREE the_grant;
}
```

The opening statement of the function that the following example shows specifies its name and its three parameters. The three parameters specify the privileges to grant, the name of the table on which to grant privileges, and the ID of the user to receive them.

```
table_grant(priv_to_grant, table_name, user_id)
char *priv_to_grant;
char *table_name;
char *user_id;
```

The function uses the statements in the following example to define a local variable, **grant_stmt**, which is used to assemble and hold the GRANT statement:

```
EXEC SQL BEGIN DECLARE SECTION;
   char grant_stmt[200];
EXEC SQL END DECLARE SECTION;
```

As the following example illustrates, the GRANT statement is created by concatenating the constant parts of the statement and the function parameters:

```
sprintf(grant_stmt, " GRANT %s ON %s TO %s",priv_to_grant,
   table_name, user_id);
```

This statement concatenates the following six character strings:

- 'GRANT'
- The parameter that specifies the privileges to be granted
- 'ON'
- The parameter that specifies the table name
- 'TO'
- The parameter that specifies the user

The result is a complete GRANT statement composed partly of program input. The PREPARE statement passes the assembled statement text to the database server for parsing.

If the database server returns an error code in SQLCODE following the PREPARE statement, the function displays an error message. If the database server approves the form of the statement, it sets a zero return code. This action does not guarantee that the statement is executed properly; it means only that the statement has correct syntax. It might refer to a nonexistent table or contain many other kinds of errors that can be detected only during execution. The following portion of the example checks that **the_grant** was prepared successfully before executing it:

```
if(SQLCODE == 0)
   EXEC SQL EXECUTE the_grant;
else
   printf("Sorry, got error # %d attempting %s", SQLCODE, grant_stmt);
```

If the preparation is successful, `SQLCODE = = 0`, the next step executes the prepared statement.

Assign roles

Alternatively, the DBA can define a role with the CREATE ROLE statement, and use the GRANT and REVOKE statements to cancel or assign roles to users, and to grant and revoke privileges of roles. For example:

```
GRANT engineer TO nmartin;
```

The SET ROLE statement is needed to activate a non-default role. For more information on roles and privileges, see Access-management strategies on page 1524 and Privileges on a database and on its objects on page 1678. For more information on the GRANT and REVOKE statements, see the *IBM® Informix® Database Design and Implementation Guide*. For more information about the syntax of these statements, see *HCL® Informix® Guide to SQL: Syntax*.

Summary

SQL statements can be written into programs as if they were normal statements of the programming language. Program variables can be used in WHERE clauses, and data from the database can be fetched into them. A preprocessor translates the SQL code into procedure calls and data structures.

Statements that do not return data, or queries that return only one row of data, are written like ordinary imperative statements of the language. Queries that can return more than one row are associated with a cursor that represents the current row of data. Through the cursor, the program can fetch each row of data as it is needed.

Static SQL statements are written into the text of the program. However, the program can form new SQL statements dynamically, as it runs, and execute them also. In the most advanced cases, the program can obtain information about the number and types of columns that a query returns and dynamically allocate the memory space to hold them.

Modify data through SQL programs

The previous chapter describes how to insert or embed SQL statements, especially the SELECT statement, into programs written in other languages. Embedded SQL enables a program to retrieve rows of data from a database.

This chapter discusses the issues that arise when a program needs to delete, insert, or update rows to modify the database. As in SQL programming on page 1700, this chapter prepares you for reading your HCL® Informix® embedded language publication.

The general use of the INSERT, UPDATE, and DELETE statements is discussed in Modify data on page 1658. This chapter examines their use from within a program. You can easily embed the statements in a program, but it can be difficult to handle errors and to deal with concurrent modifications from multiple programs.

The DELETE statement

To delete rows from a table, a program executes a DELETE statement. The DELETE statement can specify rows in the usual way, with a WHERE clause, or it can refer to a single row, the last one fetched through a specified cursor.

Whenever you delete rows, you must consider whether rows in other tables depend on the deleted rows. This problem of coordinated deletions is covered in Modify data on page 1658. The problem is the same when deletions are made from within a program.

Direct deletions

You can embed a DELETE statement in a program. The following example uses :

```
EXEC SQL delete from items
   WHERE order_num = :onum;
```

You can also prepare and execute a statement of the same form dynamically. In either case, the statement works directly on the database to affect one or more rows.

The WHERE clause in the example uses the value of a host variable named **onum**. Following the operation, results are posted in SQLSTATE and in the **sqlca** structure, as usual. The third element of the SQLERRD array contains the count of rows deleted

even if an error occurs. The value in SQLCODE shows the overall success of the operation. If the value is not negative, no errors occurred and the third element of SQLERRD is the count of all rows that satisfied the WHERE clause and were deleted.

Errors during direct deletions

When an error occurs, the statement ends prematurely. The values in SQLSTATE and in SQLCODE and the second element of SQLERRD explain its cause, and the count of rows reveals how many rows were deleted. For many errors, that count is zero because the errors prevented the database server from beginning the operation. For example, if the named table does not exist, or if a column tested in the WHERE clause is renamed, no deletions are attempted.

However, certain errors can be discovered after the operation begins and some rows are processed. The most common of these errors is a lock conflict. The database server must obtain an exclusive lock on a row before it can delete that row. Other programs might be using the rows from the table, preventing the database server from locking a row. Because the issue of locking affects all types of modifications, Programming for a multiuser environment on page 1733, discusses it.

Other, rarer types of errors can strike after deletions begin. For example, hardware errors that occur while the database is being updated.

Transaction logging

The best way to prepare for any kind of error during a modification is to use transaction logging. In the event of an error, you can tell the database server to put the database back the way it was. The following example is based on the example in the section Direct deletions on page 1723, which is extended to use transactions:

```
EXEC SQL begin work;                /* start the transaction*/
EXEC SQL delete from items
     where order_num = :onum;
del_result = sqlca.sqlcode;         /* save two error */
del_isamno = sqlca.sqlerrd[1];      /* code numbers */
del_rowcnt = sqlca.sqlerrd[2];      /* and count of rows */
if (del_result < 0)                 /* problem found: */
   EXEC SQL rollback work;          /* put everything back */
else                                /* everything worked OK:*/
   EXEC SQL commit work;            /* finish transaction */
```

A key point in this example is that the program saves the important return values in the **sqlca** structure before it ends the transaction. Both the ROLLBACK WORK and COMMIT WORK statements, like other SQL statements, set return codes in the **sqlca** structure. However, if you want to report the codes that the error generated, you must save them before executing ROLLBACK WORK. The ROLLBACK WORK statement removes all of the pending transaction, including its error codes.

The advantage of using transactions is that the database is left in a known, predictable state no matter what goes wrong. No question remains about how much of the modification is completed; either all of it or none of it is completed.

In a database with logging, if a user does not start an explicit transaction, the database server initiates an internal transaction prior to execution of the statement and terminates the transaction after execution completes or fails. If the statement execution succeeds, the internal transaction is committed. If the statement fails, the internal transaction is rolled back.

Coordinated deletions

The usefulness of transaction logging is particularly clear when you must modify more than one table. For example, consider the problem of deleting an order from the demonstration database. In the simplest form of the problem, you must delete rows from two tables, **orders** and **items**, as the following example of shows:

```
EXEC SQL BEGIN WORK;
EXEC SQL DELETE FROM items
   WHERE order_num = :o_num;
if (SQLCODE >= 0)
{
   EXEC SQL DELETE FROM orders
      WHERE order_num == :o_num;
```

```
{
   if (SQLCODE >= 0)
      EXEC SQL COMMIT WORK;
```

```
{
   else
{
      printf("Error %d on DELETE", SQLCODE);
      EXEC SQL ROLLBACK WORK;
}
```

The logic of this program is much the same whether or not transactions are used. If they are not used, the person who sees the error message has a much more difficult set of decisions to make. Depending on when the error occurred, one of the following situations applies:

- No deletions were performed; all rows with this order number remain in the database.
- Some, but not all, item rows were deleted; an order record with only some items remains.
- All item rows were deleted, but the order row remains.
- All rows were deleted.

In the second and third cases, the database is corrupted to some extent; it contains partial information that can cause some queries to produce wrong answers. You must take careful action to restore consistency to the information. When transactions are used, all these uncertainties are prevented.

Delete with a cursor

You can also write a DELETE statement with a cursor to delete the row that was last fetched. Deleting rows in this manner lets you program deletions based on conditions that cannot be tested in a WHERE clause, as the following example shows. The following example applies only to databases that are not ANSI compliant because of the way that the beginning and ending of the transaction are set up.

Warning: The design of the function in this example is unsafe. It depends on the current isolation level for correct operation. Isolation levels are discussed later in the chapter. For more information on isolation levels, see

Programming for a multiuser environment on page 1733. Even when the function works as intended, its effects depend on the physical order of rows in the table, which is not generally a good idea.

```
int delDupOrder()
{
   int ord_num;
   int dup_cnt, ret_code;

   EXEC SQL declare scan_ord cursor for
      select order_num, order_date
         into :ord_num, :ord_date
         from orders for update;
   EXEC SQL open scan_ord;
   if (sqlca.sqlcode != 0)
      return (sqlca.sqlcode);
   EXEC SQL begin work;
   for(;;)
   {
      EXEC SQL fetch next scan_ord;
      if (sqlca.sqlcode != 0) break;
      dup_cnt = 0; /* default in case of error */
      EXEC SQL select count(*) into dup_cnt from orders
         where order_num = :ord_num;
      if (dup_cnt > 1)
      {
         EXEC SQL delete from orders
            where current of scan_ord;
         if (sqlca.sqlcode != 0)
            break;
      }
   }
   ret_code = sqlca.sqlcode;
   if (ret_code == 100)             /* merely end of data */
      EXEC SQL commit work;
   else      /* error on fetch or on delete */
      EXEC SQL rollback work;
   return (ret_code);
}
```

The purpose of the function is to delete rows that contain duplicate order numbers. In fact, in the demonstration database, the **orders.order_num** column has a unique index, so duplicate rows cannot occur in it. However, a similar function can be written for another database; this one uses familiar column names.

The function declares **scan_ord**, a cursor to scan all rows in the **orders** table. It is declared with the FOR UPDATE clause, which states that the cursor can modify data. If the cursor opens properly, the function begins a transaction and then loops over rows of the table. For each row, it uses an embedded SELECT statement to determine how many rows of the table have the order number of the current row. (This step fails without the correct isolation level, as Programming for a multiuser environment on page 1733 describes.)

In the demonstration database, with its unique index on this table, the count returned to **dup_cnt** is always one. However, if it is greater, the function deletes the current row of the table, reducing the count of duplicates by one.

Cleanup functions of this sort are sometimes needed, but they generally need more sophisticated design. This function deletes all duplicate rows except the last one that the database server returns. That order has nothing to do with the content of the rows or their meanings. You can improve the function in the previous example by adding, perhaps, an ORDER BY clause to the cursor declaration. However, you cannot use ORDER BY and FOR UPDATE together. An insert example on page 1729 presents a better approach.

The INSERT statement

You can embed the INSERT statement in programs. Its form and use in a program are the same as described in Modify data on page 1658 with the additional feature that you can use host variables in expressions, both in the VALUES and WHERE clauses. Moreover, in a program you have the additional ability to insert rows with a cursor.

An insert cursor

The DECLARE CURSOR statement has many variations. Most are used to create cursors for different kinds of scans over data, but one variation creates a special kind of cursor, called an *insert cursor*. You use an insert cursor with the PUT and FLUSH statements to efficiently insert rows into a table in bulk.

Declare an insert cursor

To create an insert cursor, declare a cursor to be for an INSERT statement instead of a SELECT statement. You cannot use such a cursor to fetch rows of data; you can use it only to insert them.

When you open an insert cursor, a buffer is created in memory to hold a block of rows. The buffer receives rows of data as the program produces them; then they are passed to the database server in a block when the buffer is full. The buffer reduces the amount of communication between the program and the database server, and it lets the database server insert the rows with less difficulty. As a result, the insertions go faster.

The buffer is always made large enough to hold at least two rows of inserted values. It is large enough to hold more than two rows when the rows are shorter than the minimum buffer size.

Insert with a cursor

The code in the previous example (Declare an insert cursor on page 1727) prepares an insert cursor for use. The continuation, as the following example shows, demonstrates how the cursor can be used. For simplicity, this example assumes that a function named next_cust returns either information about a new customer or null data to signal the end of input.

```
EXEC SQL BEGIN WORK;
EXEC SQL OPEN new_custs;
while(SQLCODE == 0)
{
   next_cust();
   if(the_company == NULL)
      break;
   EXEC SQL PUT new_custs;
}
if(SQLCODE == 0)                    /* if no problem with PUT */
```

```
{
   EXEC SQL FLUSH new_custs;        /* write any rows left */
   if(SQLCODE == 0)                 /* if no problem with FLUSH */
      EXEC SQL COMMIT WORK;         /* commit changes */
}
else
   EXEC SQL ROLLBACK WORK;          /* else undo changes */
```

The code in this example calls next_cust repeatedly. When it returns non-null data, the PUT statement sends the returned data to the row buffer. When the buffer fills, the rows it contains are automatically sent to the database server. The loop normally ends when next_cust has no more data to return. Then the FLUSH statement writes any rows that remain in the buffer, after which the transaction terminates.

Re-examine the information about the INSERT statement. See The INSERT statement on page 1727. The statement by itself, not part of a cursor definition, inserts a single row into the **customer** table. In fact, the whole apparatus of the insert cursor can be dropped from the example code, and the INSERT statement can be written into the code where the PUT statement now stands. The difference is that an insert cursor causes a program to run somewhat faster.

Status codes after PUT and FLUSH

When a program executes a PUT statement, the program should test whether the row is placed in the buffer successfully. If the new row fits in the buffer, the only action of PUT is to copy the row to the buffer. No errors can occur in this case. However, if the row does not fit, the entire buffer load is passed to the database server for insertion, and an error can occur.

The values returned into the SQL Communications Area (SQLCA) give the program the information it needs to sort out each case. SQLCODE and SQLSTATE are set to zero after every PUT statement if no error occurs and to a negative error code if an error occurs.

The database server sets the third element of SQLERRD to the number of rows actually inserted into the table, as follows

- Zero, if the new row is merely moved to the buffer
- The number of rows that are in the buffer, if the buffer load is inserted without error
- The number of rows inserted before an error occurs, if one did occur

Read the code once again to see how SQLCODE is used (see the previous example). First, if the OPEN statement yields an error, the loop is not executed because the WHILE condition fails, the FLUSH operation is not performed, and the transaction rolls back. Second, if the PUT statement returns an error, the loop ends because of the WHILE condition, the FLUSH operation is not performed, and the transaction rolls back. This condition can occur only if the loop generates enough rows to fill the buffer at least once; otherwise, the PUT statement cannot generate an error.

The program might end the loop with rows still in the buffer, possibly without inserting any rows. At this point, the SQL status is zero, and the FLUSH operation occurs. If the FLUSH operation produces an error code, the transaction rolls back. Only when all inserts are successfully performed is the transaction committed.

Rows of constants

The insert cursor mechanism supports one special case where high performance is easy to obtain. In this case, all the values listed in the INSERT statement are constants: no expressions and no host variables are listed, just literal numbers and strings of characters. No matter how many times such an INSERT operation occurs, the rows it produces are identical. When the rows are identical, copying, buffering, and transmitting each identical row is pointless.

Instead, for this kind of INSERT operation, the PUT statement does nothing except to increment a counter. When a FLUSH operation is finally performed, a single copy of the row and the count of inserts are passed to the database server. The database server creates and inserts that many rows in one operation.

You do not usually insert a quantity of identical rows. You can insert identical rows when you first establish a database to populate a large table with null data.

An insert example

Delete with a cursor on page 1725 contains an example of the DELETE statement whose purpose is to look for and delete duplicate rows of a table. A better way to perform this task is to select the desired rows instead of deleting the undesired ones. The code in the following example shows one way to do this task:

```
EXEC SQL BEGIN DECLARE SECTION;
   long last_ord = 1;
   struct {
      long int o_num;
      date     o_date;
      long     c_num;
      char     o_shipinst[40];
      char     o_backlog;
      char     o_po[10];
      date     o_shipdate;
      decimal  o_shipwt;
      decimal  o_shipchg;
      date     o_paiddate;
      } ord_row;
EXEC SQL END DECLARE SECTION;

EXEC SQL BEGIN WORK;
EXEC SQL INSERT INTO new_orders
   SELECT * FROM orders main
      WHERE 1 = (SELECT COUNT(*) FROM orders minor
         WHERE main.order_num = minor.order_num);
EXEC SQL COMMIT WORK;

EXEC SQL DECLARE dup_row CURSOR FOR
   SELECT * FROM orders main INTO :ord_row
      WHERE 1 < (SELECT COUNT(*) FROM orders minor
         WHERE main.order_num = minor.order_num)
      ORDER BY order_date;
EXEC SQL DECLARE ins_row CURSOR FOR
   INSERT INTO new_orders VALUES (:ord_row);

EXEC SQL BEGIN WORK;
EXEC SQL OPEN ins_row;
```

```
EXEC SQL OPEN dup_row;
while(SQLCODE == 0)
{
   EXEC SQL FETCH dup_row;
   if(SQLCODE == 0)
   {
      if(ord_row.o_num != last_ord)
         EXEC SQL PUT ins_row;
      last_ord = ord_row.o_num
      continue;
   }
   break;
}
if(SQLCODE != 0 && SQLCODE != 100)
   EXEC SQL ROLLBACK WORK;
else
   EXEC SQL COMMIT WORK;
EXEC SQL CLOSE ins_row;
EXEC SQL CLOSE dup_row;
```

This example begins with an ordinary INSERT statement, which finds all the nonduplicated rows of the table and inserts them into another table, presumably created before the program started. That action leaves only the duplicate rows. (In the demonstration database, the **orders** table has a unique index and cannot have duplicate rows. Assume that this example deals with some other database.)

The code in the previous example then declares two cursors. The first, called **dup_row**, returns the duplicate rows in the table. Because **dup_row** is for input only, it can use the ORDER BY clause to impose some order on the duplicates other than the physical record order used in the example on page Delete with a cursor on page 1725. In this example, the duplicate rows are ordered by their dates (the oldest one remains), but you can use any other order based on the data.

The second cursor, **ins_row**, is an insert cursor. This cursor takes advantage of the ability to use a C structure, **ord_row**, to supply values for all columns in the row.

The remainder of the code examines the rows that are returned through **dup_row**. It inserts the first one from each group of duplicates into the new table and disregards the rest.

For the sake of brevity, the preceding example uses the simplest kind of error handling. If an error occurs before all rows have been processed, the sample code rolls back the active transaction.

How many rows were affected?

When your program uses a cursor to select rows, it can test SQLCODE for `100` (or SQLSTATE for `02000`), the end-of-data return code. This code is set to indicate that no rows, or no more rows, satisfy the query conditions. For databases that are not ANSI compliant, the end-of-data return code is set in SQLCODE or SQLSTATE only following SELECT statements; it is not used following DELETE, INSERT, or UPDATE statements. For ANSI-compliant databases, SQLCODE is also set to `100` for updates, deletes, and inserts that affect zero rows.

A query that finds no data is not a success. However, an UPDATE or DELETE statement that happens to update or delete no rows is still considered a success. It updated or deleted the set of rows that its WHERE clause said it should; however, the set was empty.

In the same way, the INSERT statement does not set the end-of-data return code even when the source of the inserted rows is a SELECT statement, and the SELECT statement selected no rows. The INSERT statement is a success because it inserted as many rows as it was asked to (that is, zero).

To find out how many rows are inserted, updated, or deleted, a program can test the third element of SQLERRD. The count of rows is there, regardless of the value (zero or negative) in SQLCODE.

The UPDATE statement

You can embed the UPDATE statement in a program in any of the forms that Modify data on page 1658 describes with the additional feature that you can name host variables in expressions, both in the SET and WHERE clauses. Moreover, a program can update the row that a cursor addresses.

An update cursor

An *update cursor* permits you to delete or update the current row; that is, the most recently fetched row. The following example in shows the declaration of an update cursor:

```
EXEC SQL
   DECLARE names CURSOR FOR
      SELECT fname, lname, company
      FROM customer
   FOR UPDATE;
```

The program that uses this cursor can fetch rows in the usual way.

```
EXEC SQL
   FETCH names INTO :FNAME, :LNAME, :COMPANY;
```

If the program then decides that the row needs to be changed, it can do so.

```
if (strcmp(COMPANY, "SONY") ==0)
   {
   EXEC SQL
      UPDATE customer
         SET fname = 'Midori', lname = 'Tokugawa'
         WHERE CURRENT OF names;
   }
```

The words `CURRENT OF names` take the place of the usual test expressions in the WHERE clause. In other respects, the UPDATE statement is the same as usual, even including the specification of the table name, which is implicit in the cursor name but still required.

The purpose of the keyword UPDATE

The purpose of the keyword UPDATE in a cursor is to let the database server know that the program can update (or delete) any row that it fetches. The database server places a more demanding lock on rows that are fetched through an update cursor and a less demanding lock when it fetches a row for a cursor that is not declared with that keyword. This action results in better performance for ordinary cursors and a higher level of concurrent use in a multiprocessing system. (Programming for a multiuser environment on page 1733 discusses levels of locks and concurrent use.)

Important: A normal update inside the FETCH loop of a cursor cannot guarantee that the updated rows are not fetched again after the UPDATE. The WHERE CURRENT OF specification relates the UPDATE to the Update cursor, and guarantees that each row is updated no more than once, by internally keeping a list of the rows that have already been updated. These rows will not be fetched again by the Update cursor. See the FOR UPDATE clause on page .

Update specific columns

The following example has updated specific columns of the preceding example of an update cursor:

```
EXEC SQL
   DECLARE names CURSOR FOR
      SELECT fname, lname, company, phone
         INTO  :FNAME,:LNAME,:COMPANY,:PHONE FROM customer
   FOR UPDATE OF fname, lname
END-EXEC.
```

Only the **fname** and **lname** columns can be updated through this cursor. A statement such as the following one is rejected as an error:

```
EXEC SQL
   UPDATE customer
      SET company = 'Siemens'
      WHERE CURRENT OF names
END-EXEC.
```

If the program attempts such an update, an error code is returned and no update occurs. An attempt to delete with WHERE CURRENT OF is also rejected, because deletion affects all columns.

UPDATE keyword not always needed

The ANSI standard for SQL does not provide for the FOR UPDATE clause in a cursor definition. When a program uses an ANSI-compliant database, it can update or delete with any cursor.

Cleanup a table

A final, hypothetical example of how to use an update cursor presents a problem that should never arise with an established database but could arise in the initial design phases of an application.

In the example, a large table named **target** is created and populated. A character column, **dactyl**, inadvertently acquires some null values. These rows should be deleted. Furthermore, a new column, **serials**, is added to the table with the ALTER TABLE statement. This column is to have unique integer values installed. The following example shows the code you use to accomplish these tasks:

```
EXEC SQL BEGIN DECLARE SECTION;
char dcol[80];
short dcolint;
int sequence;
EXEC SQL END DECLARE SECTION;
```

```
EXEC SQL DECLARE target_row CURSOR FOR
   SELECT datcol
      INTO :dcol:dcolint
      FROM target
   FOR UPDATE OF serials;
EXEC SQL BEGIN WORK;
EXEC SQL OPEN target_row;
if (sqlca.sqlcode == 0) EXEC SQL FETCH NEXT target_row;
for(sequence = 1; sqlca.sqlcode == 0; ++sequence)
{
   if (dcolint < 0) /* null datcol */
      EXEC SQL DELETE WHERE CURRENT OF target_row;
   else
      EXEC SQL UPDATE target SET serials = :sequence
         WHERE CURRENT OF target_row;
}
if (sqlca.sqlcode >= 0)
   EXEC SQL COMMIT WORK;
else EXEC SQL ROLLBACK WORK;
```

Summary

A program can execute the INSERT, DELETE, and UPDATE statements, as Modify data on page 1658 describes. A program can also scan through a table with a cursor, updating or deleting selected rows. It can also use a cursor to insert rows, with the benefit that the rows are buffered and sent to the database server in blocks.

In all these activities, you must make sure that the program detects errors and returns the database to a known state when an error occurs. The most important tool for doing this is transaction logging. Without transaction logging, it is more difficult to write programs that can recover from errors.

Programming for a multiuser environment

This section describes several programming issues you need to be aware of when you work in a multiuser environment.

If your database is contained in a single-user workstation and does not access data from another computer, your programs can modify data freely. In all other cases, you must allow for the possibility that, while your program is modifying data, another program is reading or modifying the same data. This situation is described as *concurrency*: two or more independent uses of the same data at the same time. This section addresses concurrency, locking, and isolation levels.

This section also describes the statement cache feature, which can reduce per-session memory allocation and speed up query processing. The statement cache stores statements that can then be shared among different user sessions that use identical SQL statements.

Concurrency and performance

Concurrency is crucial to good performance in a multiprogramming system. When access to the data is *serialized* so that only one program at a time can use it, processing slows dramatically.

Locks and integrity

Unless controls are placed on the use of data, concurrency can lead to a variety of negative effects. Programs can read obsolete data, or modifications can be lost even though they were apparently completed.

To prevent errors of this kind, the database server imposes a system of *locks*. A lock is a claim, or reservation, that a program can place on a piece of data. The database server guarantees that, as long as the data is locked, no other program can modify it. When another program requests the data, the database server either makes the program wait or turns it back with an error.

Locks and performance

Because a lock serializes access to one piece of data, it reduces concurrency; any other programs that want access to that data must wait. The database server can place a lock on a single row, a disk page, a whole table, or an entire database. (A disk page might hold multiple rows and a row might require multiple disk pages.) The more locks it places and the larger the objects it locks, the more concurrency is reduced. The fewer the locks and the smaller the locked objects, the greater concurrency and performance can be.

The following sections discuss how you can achieve the following goals with your program:

- Place all the locks necessary to ensure data integrity.
- Lock the fewest, smallest pieces of data possible consistent with the preceding goal.

Concurrency issues

To understand the hazards of concurrency, you must think in terms of multiple programs, each executing at its own speed. Suppose that your program is fetching rows through the following cursor:

```
EXEC SQL DECLARE sto_curse CURSOR FOR
   SELECT * FROM stock
      WHERE manu_code = 'ANZ';
```

The transfer of each row from the database server to the program takes time. During and between transfers, other programs can perform other database operations. At about the same time that your program fetches the rows produced by that query, another user's program might execute the following update:

```
EXEC SQL UPDATE stock
   SET unit_price = 1.15 * unit_price
      WHERE manu_code = 'ANZ';
```

In other words, both programs are reading through the same table, one fetching certain rows and the other changing the same rows. The following scenarios are possible:

1. The other program finishes its update before your program fetches its first row.

   Your program shows you only updated rows.
2. Your program fetches every row before the other program has a chance to update it.

   Your program shows you only original rows.

3. After your program fetches some original rows, the other program catches up and goes on to update some rows that your program has yet to read; then it executes the COMMIT WORK statement.

   Your program might return a mixture of original rows and updated rows.

4. Same as number 3, except that after updating the table, the other program issues a ROLLBACK WORK statement.

   Your program can show you a mixture of original rows and updated rows that no longer exist in the database.

The first two possibilities are harmless. In possibility number 1, the update is complete before your query begins. It makes no difference whether the update finished a microsecond ago or a week ago.

In possibility number 2, your query is, in effect, complete before the update begins. The other program might have been working just one row behind yours, or it might not start until tomorrow night; it does not matter.

The last two possibilities, however, can be important to the design of some applications. In possibility number 3, the query returns a mix of updated and original data. That result can be detrimental in some applications. In others, such as one that is taking an average of all prices, it might not matter at all.

Possibility number 4 can be disastrous if a program returns some rows of data that, because their transaction was cancelled, can no longer be found in the table.

Another concern arises when your program uses a cursor to update or delete the last-fetched row. Erroneous results occur with the following sequence of events:

- Your program fetches the row.
- Another program updates or deletes the row.
- Your program updates or deletes WHERE CURRENT OF *cursor_name*.

To control concurrent events such as these, use the locking and *isolation level* features of the database server.

How locks work

HCL® Informix® database servers support a complex, flexible set of locking features that the topics in this section describe.

Kinds of locks

The following table shows the types of locks that HCL® Informix® database servers support for different situations.

Lock type	Use
Shared	A shared lock reserves its object for reading only. It prevents the object from changing while the lock remains. More than one program can place a shared lock on the same object. More than one object can read the record while it is locked in shared mode.

Lock type	Use
Exclusive	An exclusive lock reserves its object for the use of a single program. This lock is used when the program intends to change the object. You cannot place an exclusive lock where any other kind of lock exists. After you place an exclusive lock, you cannot place another lock on the same object.
Promotable (or Update)	A promotable (or update) lock establishes the intent to update. You can only place it where no other promotable or exclusive lock exists. You can place promotable locks on records that already have shared locks. When the program is about to change the locked object, you can promote the promotable lock to an exclusive lock, but only if no other locks, including shared locks, are on the record at the time the lock would change from promotable to exclusive. If a shared lock was on the record when the promotable lock was set, you must drop the shared lock before the promotable lock can be promoted to an exclusive lock.

Lock scope

You can apply locks to entire databases, entire tables, disk pages, single rows, or index-key values. The size of the object that is being locked is referred to as the *scope* of the lock (also called the *lock granularity*). In general, the larger the scope of a lock, the more concurrency is reduced, but the simpler programming becomes.

Database locks

You can lock an entire database. The act of opening a database places a shared lock on the name of the database. A database is opened with the CONNECT, DATABASE, or CREATE DATABASE statements. As long as a program has a database open, the shared lock on the name prevents any other program from dropping the database or putting an exclusive lock on it.

The following statement shows how you might lock an entire database exclusively:

```
DATABASE database_one EXCLUSIVE
```

This statement succeeds if no other program has opened that database. After the lock is placed, no other program can open the database, even for reading, because its attempt to place a shared lock on the database name fails.

A database lock is released only when the database closes. That action can be performed explicitly with the DISCONNECT or CLOSE DATABASE statements or implicitly by executing another DATABASE statement.

Because locking a database reduces concurrency in that database to zero, it makes programming simple; concurrent effects cannot happen. However, you should lock a database only when no other programs need access. Database locking is often used before applying massive changes to data during off-peak hours.

Table locks

You can lock entire tables. In some cases, the database server performs this action automatically. You can also use the LOCK TABLE statement to lock an entire table explicitly.

The LOCK TABLE statement or the database server can place the following types of table locks:

Shared lock

No users can write to the table. In shared mode, the database server places one shared lock on the table, which informs other users that no updates can be performed. In addition, the database server adds locks for every row updated, deleted, or inserted.

Exclusive lock

No other users can read from or write to the table. In exclusive mode, the database server places only one exclusive lock on the table, no matter how many rows it updates. An exclusive table lock prevents any concurrent use of the table and, therefore, can have a serious effect on performance if many other programs are contending for the use of the table. However, when you need to update most of the rows in a table, place an exclusive lock on the table.

Lock a table with the LOCK TABLE statement

A transaction tells the database server to use table-level locking for a table with the LOCK TABLE statement. The following example shows how to place an exclusive lock on a table:

```
LOCK TABLE tab1 IN EXCLUSIVE MODE
```

The following example shows how to place a shared lock on a table:

```
LOCK TABLE tab2 IN SHARE MODE
```

Tip: You can set the isolation level for your database server to achieve the same degree of protection as the shared table lock while providing greater concurrency.

When the database server automatically locks a table

The database server always locks an entire table while it performs operations for any of the following statements:

- ALTER FRAGMENT
- ALTER INDEX
- ALTER TABLE
- CREATE INDEX
- DROP INDEX
- RENAME COLUMN
- RENAME TABLE

Completion of the statement (or end of the transaction) releases the lock. An entire table can also be locked automatically during certain queries.

Avoid table locking with the ONLINE keyword

For indexes that are not defined with the IN TABLE keyword option, you can minimize the duration of an exclusive lock on the indexed table when you CREATE or DROP an index using the ONLINE keyword.

While the index is being created or dropped online, no DDL operations on the table are supported, but operations that were concurrent when the CREATE INDEX or DROP INDEX statement was issued can be completed. The specified index is not created or dropped until no other processes are concurrently accessing the table. Then locks are held briefly to write the system catalog data associated with the index. This increases the availability of the system, because the table is still readable by ongoing and new sessions. The following statement shows how to use the ONLINE keyword to avoid automatic table locking with a CREATE INDEX statement:

```
CREATE INDEX idx_1 ON customer (lname) ONLINE;
```

For in-table indexes, however, that are defined with the IN TABLE keyword option, the indexed table remains locked for the duration of the CREATE INDEX or DROP INDEX operation that includes the ONLINE keyword. Attempted access by other sessions to the locked table would fail with at least one of these errors:

```
107: ISAM error:  record is locked.
211: Cannot read system catalog (systables).
710: Table (table.tix) has been dropped, altered or renamed.
```

Row and key locks

You can lock one row of a table. A program can lock one row or a selection of rows while other programs continue to work on other rows of the same table.

Row and key locking are not the default behaviors. You must specify row-level locking when you create the table. The following example creates a table with row-level locking:

```
CREATE TABLE tab1
(
col1...
) LOCK MODE ROW;
```

If you specify a LOCK MODE clause when you create a table, you can later change the lock mode with the ALTER TABLE statement. The following statement changes the lock mode on the reservations table to page-level locking:

```
ALTER TABLE tab1 LOCK MODE PAGE
```

In certain cases, the database server has to lock a row that does not exist. To do this, the database server places a lock on an index-key value. Key locks are used identically to row locks. When the table uses row locking, key locks are implemented as locks on imaginary rows. When the table uses page locking, a key lock is placed on the index page that contains the key or that would contain the key if it existed.

When you insert, update, or delete a key (performed automatically when you insert, update, or delete a row), the database server creates a lock on the key in the index.

Row and key locks generally provide the best performance overall when you update a relatively small number of rows because they increase concurrency. However, the database server incurs some overhead in obtaining a lock.

When one or more rows in a table are locked by an exclusive lock, the effect on other users partly depends on their transaction isolation level. Other users whose isolation levels is not Dirty Read might encounter transactions that fail because the exclusive lock was not released within a specified time limit.

For Committed Read or Dirty Read isolation level operations that attempt to access tables on which a concurrent session has set exclusive row-level locks, the risk of locking conflicts can be reduced by enabling transactions to read the most recently committed version of the data in the locked rows, rather than waiting for the transaction that set the lock to be committed or rolled back. Enabling access to the last committed version of exclusively locked rows can be accomplished in several ways:

- For an individual session, issue this SQL statement
  ```
  SET ISOLATION TO COMMITTED READ LAST COMMITTED;
  ```
- For all sessions using the Committed Read or Read Committed isolation level, the DBA can set the USELASTCOMMITTED configuration parameter to `'ALL'` or to `'COMMITTED READ'`.
- For an individual session using the Committed Read or Read Committed isolation level, any user can issue the SET ENVIRONMENT USELASTCOMMITTED statement with `'ALL'` or `'COMMITTED READ'` as the value of this session environment option.
- For all sessions using Dirty Read or Read Uncommitted isolation levels, the DBA can set the USELASTCOMMITTED configuration parameter to `'ALL'` or to `'DIRTY READ'`.
- For an individual session using the Dirty Read or Read Uncommitted isolation levels, any user can issue the SET ENVIRONMENT USELASTCOMMITTED statement with `'ALL'` or `'DIRTY READ'` as the value of this session environment option.

This LAST COMMITTED feature is useful only when row-level locking is in effect, rather than when another session holds an exclusive lock on the entire table. This feature is disabled for any table on which the LOCK TABLE statement applies a table-level lock. See the description of the SET ENVIRONMENT statement in the *HCL® Informix® Guide to SQL: Syntax* and the description of the USELASTCOMMITTED configuration parameter in*HCL® Informix® Administrator's Reference* for more information about this feature for concurrent access to tables in which some rows are locked by exclusive locks, and for restrictions on the kinds of tables that can support this feature.

Page locks

The database server stores data in units called *disk pages*. A disk page contains one or more rows. In some cases, it is better to lock a disk page than to lock individual rows on it. For example, with operations that require changing a large number of rows, you might choose page-level locking because row-level locking (one lock per row) might not be cost effective.

If you do not specify a LOCK MODE clause when you create a table, the default behavior for the database server is page-level locking. With page locking, the database server locks the entire page that contains the row. If you update several rows that are stored on the same page, the database server uses only one lock for the page.

Set the row or page lock mode for all CREATE TABLE statements

HCL Informix® allows you to set the lock mode to page-level locking or row-level locking for all newly created tables for a single user (per session) or for multiple users (per server). You no longer need to specify the lock mode every time that you create a new table with the CREATE TABLE statement.

If you want every new table created within your session to be created with a particular lock mode, you have to set the **IFX_DEF_TABLE_LOCKMODE** environment variable. For example, for every new table created within your session to be created with lock mode row, set **IFX_DEF_TABLE_LOCKMODE** to ROW. To override this behavior, use the CREATE TABLE or ALTER TABLE statements and redefine the LOCK MODE clause.

Single-user lock mode

Set the single-user lock mode if all of the new tables that you create in your session require the same lock mode. Set the single-user lock mode with the **IFX_DEF_TABLE_LOCKMODE** environment variable. For example, for every new table created within your session to be created with row-level locking, set **IFX_DEF_TABLE_LOCKMODE** to ROW. To override this behavior, use the CREATE TABLE or ALTER TABLE statements and redefine the LOCK MODE clause. For more information on setting environment variables, see the *HCL® Informix® Guide to SQL: Reference*.

Multiple-user lock mode

Database administrators can use the multiple-user lock mode to create greater concurrency by designating the lock mode for all users on the same server. All tables that any user creates on that server will then have the same lock mode. To enable multiple-user lock mode, set the **IFX_DEF_TABLE_LOCKMODE** environment variable before starting the database server or set the DEF_TABLES_LOCKMODE configuration parameter.

Rules of precedence

Locking mode for CREATE TABLE or ALTER TABLE has the following rules of precedence, listed in order of highest precedence to lowest:

1. CREATE TABLE or ALTER TABLE SQL statements that use the LOCK MODE clause
2. Single-user environment variable setting
3. Multi-user environment variable setting in the server environment
4. Configuration parameters in the configuration file
5. Default behavior (page-level locking)

Coarse index locks

When you change the lock mode of an index from normal to coarse lock mode, index-level locks are acquired on the index instead of item-level or page-level locks, which are the normal locks. This mode reduces the number of lock calls on an index.

Use the coarse lock mode when you know the index is not going to change; that is, when read-only operations are performed on the index.

Use the normal lock mode to have the database server place item-level or page-level locks on the index as necessary. Use this mode when the index gets updated frequently.

When the database server executes the command to change the lock mode to coarse, it acquires an exclusive lock on the table for the duration of the command. Any transactions that are currently using a lock of finer granularity must complete before the database server switches to the coarse lock mode.

Smart-large-object locks

Locks on a CLOB or BLOB column are separate from the lock on the row. Smart large objects are locked only when they are accessed. When you lock a table that contains a CLOB or BLOB column, no smart large objects are locked. If accessed for writing, the smart large object is locked in update mode, and the lock is promoted to exclusive when the actual write occurs. If accessed for reading, the smart large object is locked in shared mode. The database server recognizes the transaction isolation mode, so if Repeatable Read isolation level is set, the database server does not release smart-large-object read locks before end of transaction.

When the database server retrieves a row and updates a smart large object that the row points to, only the smart large object is exclusively locked during the time it is being updated.

Byte-range locks

You can lock a range of bytes for a smart large object. Byte-range locks allow a transaction to selectively lock only those bytes that are accessed so that writers and readers simultaneously can access different byte ranges in the same smart large object.

For information about how to use byte-range locks, see your .

Byte-range locks support deadlock detection. For information about deadlock detection, see Handle a deadlock on page 1750.

Duration of a lock

The program controls the duration of a database lock. A database lock is released when the database closes.

Depending on whether the database uses transactions, table lock durations vary. If the database does not use transactions (that is, if no transaction log exists and you do not use a COMMIT WORK statement), a table lock remains until it is removed by the execution of the UNLOCK TABLE statement.

The duration of table, row, and index locks depends on what SQL statements you use and on whether transactions are in use.

When you use transactions, the end of a transaction releases all table, row, page, and index locks. When a transaction ends, all locks are released.

Locks while modifying

When the database server fetches a row through an update cursor, it places a promotable lock on the fetched row. If this action succeeds, the database server knows that no other program can alter that row. Because a promotable lock is not

exclusive, other programs can continue to read the row. A promotable lock can improve performance because the program that fetched the row can take some time before it issues the UPDATE or DELETE statement, or it can simply fetch the next row. When it is time to modify a row, the database server obtains an exclusive lock on the row. If it already has a promotable lock, it changes that lock to exclusive status.

The duration of an exclusive row lock depends on whether transactions are in use. If they are not in use, the lock is released as soon as the modified row is written to disk. When transactions are in use, all such locks are held until the end of the transaction. This action prevents other programs from using rows that might be rolled back to their original state.

When transactions are in use, a key lock is used whenever a row is deleted. Using a key lock prevents the following error from occurring:

- Program A deletes a row.
- Program B inserts a row that has the same key.
- Program A rolls back its transaction, forcing the database server to restore its deleted row.

  What is to be done with the row inserted by Program B?

By locking the index, the database server prevents a second program from inserting a row until the first program commits its transaction.

The locks placed while the database server reads various rows are controlled by the current isolation level, as discussed in the next section.

Lock with the SELECT statement

The type and duration of locks that the database server places depend on the isolation set in the application and whether the SELECT statement is within an update cursor.

This section describes the different isolation levels and update cursors.

Set the isolation level

The *isolation level* is the degree to which your program is isolated from the concurrent actions of other programs. The database server offers a choice of isolation levels that reflect a different set of rules for how a program uses locks when it reads data.

To set the isolation level, use either the SET ISOLATION or SET TRANSACTION statement. The SET TRANSACTION statement also lets you set access modes. For more information about access modes, see Control data modification with access modes on page 1749.

SET TRANSACTION versus SET ISOLATION

The SET TRANSACTION statement complies with ANSI SQL-92. This statement is similar to the HCL® Informix® SET ISOLATION statement; however, the SET ISOLATION statement is not ANSI compliant and does not provide access modes.

The following table shows the relationships between the isolation levels that you set with the SET TRANSACTION and SET ISOLATION statements.

SET TRANSACTION correlates with	SET ISOLATION
Read Uncommitted	Dirty Read
Read Committed	Committed Read
Not Supported	Cursor Stability
(ANSI) Repeatable Read	(HCL® Informix®) Repeatable Read
Serializable	(HCL® Informix®) Repeatable Read

The major difference between the SET TRANSACTION and SET ISOLATION statements is the behavior of the isolation levels within transactions. The SET TRANSACTION statement can be issued only once for a transaction. Any cursors opened during that transaction are guaranteed to have that isolation level (or access mode if you are defining an access mode). With the SET ISOLATION statement, after a transaction is started, you can change the isolation level more than once within the transaction. The following examples illustrate the difference between the use of SET ISOLATION and the use of SET TRANSACTION.

SET ISOLATION

```
EXEC SQL BEGIN WORK;
EXEC SQL SET ISOLATION TO DIRTY READ;
EXEC SQL SELECT ... ;
EXEC SQL SET ISOLATION TO REPEATABLE READ;
EXEC SQL INSERT ... ;
EXEC SQL COMMIT WORK;
   -- Executes without error
```

SET TRANSACTION

```
EXEC SQL BEGIN WORK;
EXEC SQL SET TRANSACTION ISOLATION LEVEL TO SERIALIZABLE;
EXEC SQL SELECT ... ;
EXEC SQL SET TRANSACTION ISOLATION LEVEL TO READ COMMITTED;
Error -876: Cannot issue SET TRANSACTION once a transaction has started.
```

ANSI Read Uncommitted and HCL Informix® Dirty Read isolation

The simplest isolation level, ANSI Read Uncommitted and HCL® Informix® Dirty Read, amounts to virtually no isolation. When a program fetches a row, it places no locks, and it respects none; it simply copies rows from the database without regard to what other programs are doing.

A program always receives complete rows of data. Even under ANSI Read Uncommitted or HCL® Informix® Dirty Read isolation, a program never sees a row in which some columns are updated and some are not. However, a program that uses ANSI Read Uncommitted or HCL® Informix® Dirty Read isolation sometimes reads updated rows before the updating

program ends its transaction. If the updating program later rolls back its transaction, the reading program processes data that never really existed (possibility number 4 on page 1735 in the list of concurrency issues).

ANSI Read Uncommitted or HCL® Informix® Dirty Read is the most efficient isolation level. The reading program never waits and never makes another program wait. It is the preferred level in any of the following cases:

- All tables are static; that is, concurrent programs only read and never modify data.
- The table is held in an exclusive lock.
- Only one program is using the table.

ANSI Read Committed and HCL Informix® Committed Read isolation

When a program requests the ANSI Read Committed or HCL Informix® Committed Read isolation level, the database server guarantees that it never returns a row that is not committed to the database. This action prevents reading data that is not committed and that is subsequently rolled back.

ANSI Read Committed or HCL Informix® Committed Read is implemented simply. Before it fetches a row, the database server tests to determine whether an updating process placed a lock on the row; if not, it returns the row. Because rows that have been updated (but that are not yet committed) have locks on them, this test ensures that the program does not read uncommitted data.

ANSI Read Committed or HCL Informix® Committed Read does not actually place a lock on the fetched row, so this isolation level is almost as efficient as ANSI Read Uncommitted or HCL Informix® Dirty Read. This isolation level is appropriate to use when each row of data is processed as an independent unit, without reference to other rows in the same or other tables.

Locking conflicts can occur in ANSI Read Committed or HCL Informix® Committed Read sessions, however, if the attempt to place the test lock is not successful because a concurrent session holds a shared lock on the row. To avoid waiting for concurrent transactions to release shared locks (by committing or rolling back), Informix® supports the Last Committed option to the Committed Read isolation level. When this Last Committed option is in effect, a shared lock by another session causes the query to return the most recently committed version of the row.

The Last Committed feature can also be activated by setting the USELASTCOMMITTED configuration parameter to `'COMMITTED READ'` or to `'ALL'`, or by setting USELASTCOMMITTED session environment option in the SET ENVIRONMENT statement in the **sysdbopen( )** procedure when the user connects to the database. For more information about the Last Committed option to the ANSI Read Committed or HCL Informix® Committed Read isolation levels, see the description of the SET ISOLATION statement in the *HCL® Informix® Guide to SQL: Syntax*. For information about the USELASTCOMMITTED configuration parameter, see the *HCL® Informix® Administrator's Reference*.

HCL Informix® Cursor Stability isolation

The next level, Cursor Stability, is available only with the Informix® SQL statement SET ISOLATION.

When Cursor Stability is in effect, Informix® places a lock on the latest row fetched. It places a shared lock for an ordinary cursor or a promotable lock for an update cursor. Only one row is locked at a time; that is, each time a row is fetched, the lock on the previous row is released (unless that row is updated, in which case the lock holds until the end of the

transaction). Because Cursor Stability locks only one row at a time, it restricts concurrency less than a table lock or database lock.

Cursor Stability ensures that a row does not change while the program examines it. Such row stability is important when the program updates some other table based on the data it reads from the row. Because of Cursor Stability, the program is assured that the update is based on current information. It prevents the use of *stale data*.

The following example illustrates effective use of Cursor Stability isolation. In terms of the demonstration database, Program A wants to insert a new stock item for manufacturer Hero (HRO). Concurrently, Program B wants to delete manufacturer HRO and all stock associated with it. The following sequence of events can occur:

1. Program A, operating under Cursor Stability, fetches the HRO row from the **manufact** table to learn the manufacturer code. This action places a shared lock on the row.
2. Program B issues a DELETE statement for that row. Because of the lock, the database server makes the program wait.
3. Program A inserts a new row in the **stock** table using the manufacturer code it obtained from the **manufact** table.
4. Program A closes its cursor on the **manufact** table or reads a different row of it, releasing its lock.
5. Program B, released from its wait, completes the deletion of the row and goes on to delete the rows of **stock** that use manufacturer code HRO, including the row that Program A just inserted.

If Program A used a lesser level of isolation, the following sequence could occur:

1. Program A reads the HRO row of the **manufact** table to learn the manufacturer code. No lock is placed.
2. Program B issues a DELETE statement for that row. It succeeds.
3. Program B deletes all rows of **stock** that use manufacturer code HRO.
4. Program B ends.
5. Program A, not aware that its copy of the HRO row is now invalid, inserts a new row of **stock** using the manufacturer code HRO.
6. Program A ends.

At the end, a row occurs in **stock** that has no matching manufacturer code in **manufact**. Furthermore, Program B apparently has a bug; it did not delete the rows that it was supposed to delete. Use of the Cursor Stability isolation level prevents these effects.

The preceding scenario could be rearranged to fail even with Cursor Stability. All that is required is for Program B to operate on tables in the reverse sequence to Program A. If Program B deletes from **stock** before it removes the row of **manufact**, no degree of isolation can prevent an error. Whenever this kind of error is possible, all programs that are involved must use the same sequence of access.

ANSI Serializable, ANSI Repeatable Read, and HCL Informix® Repeatable Read isolation

Where ANSI Serializable or ANSI Repeatable Read are required, a single isolation level is provided, called HCL Informix® Repeatable Read. This is logically equivalent to ANSI Serializable. Because ANSI Serializable is more restrictive than ANSI

Repeatable Read, HCL Informix® Repeatable Read can be used when ANSI Repeatable Read is desired (although HCL Informix® Repeatable Read is more restrictive than is necessary in such contexts).

The Repeatable Read isolation level asks the database server to put a lock on every row the program examines and fetches. The locks that are placed are shareable for an ordinary cursor and promotable for an update cursor. The locks are placed individually as each row is examined. They are not released until the cursor closes or a transaction ends.

Repeatable Read allows a program that uses a scroll cursor to read selected rows more than once and to be sure that they are not modified or deleted between readings. (SQL programming on page 1700 describes scroll cursors.) No lower isolation level guarantees that rows still exist and are unchanged the second time they are read.

Repeatable Read isolation places the largest number of locks and holds them the longest. Therefore, it is the level that reduces concurrency the most. If your program uses this level of isolation, think carefully about how many locks it places, how long they are held, and what the effect can be on other programs.

In addition to the effect on concurrency, the large number of locks can be a problem. The database server records the number of locks by each program in a lock table. If the maximum number of locks is exceeded, the lock table fills up, and the database server cannot place a lock. An error code is returned. The person who administers the Informix® database server system can monitor the lock table and tell you when it is heavily used.

The isolation level in an ANSI-compliant database is set to Serializable by default. The Serializable isolation level is required to ensure that operations behave according to the ANSI standard for SQL.

Update cursors

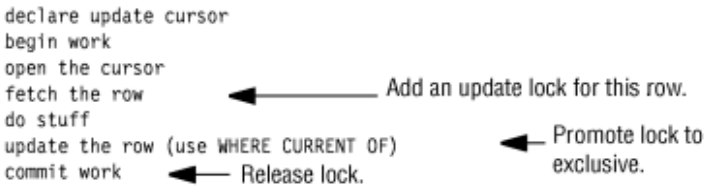
An update cursor is a special kind of cursor that applications can use when the row might potentially be updated. To use an update cursor, execute SELECT FOR UPDATE in your application. Update cursors use *promotable locks*; that is, the database server places an update lock (meaning other users can still view the row) when the application fetches the row, but the lock is changed to an exclusive lock when the application updates the row using an update cursor and UPDATE...WHERE CURRENT OF.

The advantage of using an update cursor is that you can view the row with the confidence that other users cannot change it or view it with an update cursor while you are viewing it and before you update it.

Tip: In an ANSI-compliant database, update cursors are unnecessary because any select cursor behaves the same as an update cursor.

The pseudocode in the following figure shows when the database server places and releases locks with a cursor.

Figure 382. Locks Placed for Update



Retain update locks

If a user has the isolation level set lower than Repeatable Read, the database server releases update locks placed on rows as soon as the next row is fetched from a cursor. With this feature, you can use the RETAIN UPDATE LOCKS clause to retain an update lock until the end of a transaction when you set any of the following isolation levels:

- Dirty Read
- Committed Read
- Cursor Stability

This feature lets you avoid the overhead of Repeatable Read isolation level or workarounds such as dummy updates on a row. When the RETAIN UPDATE LOCKS feature is turned on and an update lock is implicitly placed on a row during a fetch of a SELECT...FOR UPDATE statement, the update lock is not released until the end of the transaction. With the RETAIN UPDATE LOCKS feature, only update locks are held until end of transaction, whereas the Repeatable Read isolation level holds both update locks and shared locks until end of transaction.

The following example shows how to use the RETAIN UPDATE LOCKS clause when you set the isolation level to Committed Read.

```
SET ISOLATION TO COMMITTED READ RETAIN UPDATE LOCKS
```

To turn off the RETAIN UPDATE LOCKS feature, set the isolation level without the RETAIN UPDATE LOCKS clause. When you turn off the feature, update locks are not released directly. However, from this point on, a subsequent fetch releases the update lock of the immediately preceding fetch but not of earlier fetch operations. A close cursor releases the update lock on the current row.

For more information about how to use the RETAIN UPDATE LOCKS feature when you specify an isolation level, see the *HCL® Informix® Guide to SQL: Syntax*.

Exclusive locks that occur with some SQL statements

When you execute an INSERT, UPDATE, or DELETE statement, the database server uses *exclusive locks*. An exclusive lock means that no other users can update or delete the item until the database server removes the lock. In addition, no other users can view the row unless they are using the Dirty Read isolation level.

When the database server removes the exclusive lock depends on whether the database supports transaction logging.

For more information about these exclusive locks, see Locks placed with INSERT, UPDATE, and DELETE statements on page .

The behavior of the lock types

HCL® Informix® database servers store locks in an internal lock table. When the database server reads a row, it checks if the row or its associated page, table, or database is listed in the lock table. If it is in the lock table, the database server must also check the lock type. The lock table can contain the following types of locks.

Lock name	Description	Statement usually placing the lock
S	Shared lock	SELECT
X	Exclusive lock	INSERT, UPDATE, DELETE
U	Update lock	SELECT in an update cursor
B	Byte lock	Any statement that updates VARCHAR columns

In addition, the lock table might store *intent locks*. An intent lock can be an intent shared (IS), intent exclusive (IX), or intent shared exclusive (SIX). An intent lock is the lock the database server (lock manager) places on a higher granularity object when a lower granularity object needs to be locked. For example, when a user locks a row or page in Shared lock mode, the database server places an IS (intent shared) lock on the table to provide an instant check that no other user holds an X lock on the table. In this case, intent locks are placed on the table only and not on the row or page. Intent locks can be placed at the level of a row, page, or table only.

The user does not have direct control over intent locks; the lock manager internally manages all intent locks.

The following table shows what locks a user (or the database server) can place if another user (or the database server) holds a certain type of lock. For example, if one user holds an exclusive lock on an item, another user requesting any kind of lock (exclusive, update or shared) receives an error. In addition, the database server is unable to place any intent locks on an item if a user holds an exclusive lock on the item.

	Hold X lock	Hold U lock	Hold S lock	Hold IS lock	Hold SIX lock	Hold IX lock
Request X lock	No	No	No	No	No	No
Request U lock	No	No	Yes	Yes	No	No
Request S lock	No	Yes	Yes	Yes	No	No
Request IS lock	No	Yes	Yes	Yes	Yes	Yes
Request SIX lock	No	No	No	Yes	No	No
Request IX lock	No	No	No	Yes	No	Yes

For information about how locking affects performance, see your .

Control data modification with access modes

HCL® Informix® database servers support access modes. Access modes affect read and write concurrency for rows within transactions and are set with the SET TRANSACTION statement. You can use access modes to control data modification among shared files.

Transactions are read-write by default. If you specify that a transaction is read-only, that transaction cannot perform the following tasks:

- Insert, delete, or update table rows
- Create, alter, or drop any database object such as a schema, table, temporary table, index, or stored routine
- Grant or revoke privileges
- Update statistics
- Rename columns or tables

Read-only access mode prohibits updates.

You can execute stored routines in a read-only transaction as long as the routine does not try to perform any restricted operations.

For information about how to use the SET TRANSACTION statement to specify an access mode, see the *HCL® Informix® Guide to SQL: Syntax*.

Set the lock mode

The lock mode determines what happens when your program encounters locked data. One of the following situations occurs when a program attempts to fetch or modify a locked row:

- The database server immediately returns an error code in SQLCODE or SQLSTATE to the program.
- The database server suspends the program until the program that placed the lock removes the lock.
- The database server suspends the program for a time and then, if the lock is not removed, the database server sends an error-return code to the program.

You choose among these results with the SET LOCK MODE statement.

Waiting for locks

When a user encounters a lock, the default behavior of a database server is to return an error to the application. If you prefer to wait indefinitely for a lock (this choice is best for many applications), you can execute the following SQL statement:

```
SET LOCK MODE TO WAIT
```

When this lock mode is set, your program usually ignores the existence of other concurrent programs. When your program needs to access a row that another program has locked, it waits until the lock is removed, then proceeds. In most cases, the delays are imperceptible.

You can also wait for a specific number of seconds, as in the following example:

```
SET LOCK MODE TO WAIT 20
```

Not waiting for locks

The disadvantage of waiting for locks is that the wait might become long (although properly designed applications should hold their locks briefly). When the possibility of a long delay is not acceptable, a program can execute the following statement:

```
SET LOCK MODE TO NOT WAIT
```

When the program requests a locked row, it immediately receives an error code (for example, error -107 `Record is locked`), and the current SQL statement terminates. The program must roll back its current transaction and try again.

The initial setting is not waiting when a program starts up. If you are using SQL interactively and see an error related to locking, set the lock mode to wait. If you are writing a program, consider making that one of the first embedded SQL statements that the program executes.

Limited time wait

You can ask the database server to set an upper limit on a wait with the following statement:

```
SET LOCK MODE TO WAIT 17
```

This statement places an upper limit of 17 seconds on the length of any wait. If a lock is not removed in that time, the error code is returned.

Handle a deadlock

A *deadlock* is a situation in which a pair of programs blocks the progress of each other. Each program has a lock on some object that the other program wants to access. A deadlock arises only when all programs concerned set their lock modes to wait for locks.

The HCL® Informix® database server detects deadlocks immediately when they only involve data at a single network server. It prevents the deadlock from occurring by returning an error code (error -143 `ISAM error: deadlock detected`) to the second program to request a lock. The error code is the one the program receives if it sets its lock mode to not wait for locks. If your program receives an error code related to locks even after it sets lock mode to wait, you know the cause is an impending deadlock.

Handling external deadlock

A deadlock can also occur between programs on different database servers. In this case, the database server cannot instantly detect the deadlock. (Perfect deadlock detection requires excessive communications traffic among all database servers in a network.) Instead, each database server sets an upper limit on the amount of time that a program can wait to obtain a lock on data at a different database server. If the time expires, the database server assumes that a deadlock was the cause and returns a lock-related error code.

In other words, when external databases are involved, every program runs with a maximum lock-waiting time. The DBA can set or modify the maximum for the database server.

Simple concurrency

If you are not sure which choice to make concerning locking and concurrency, you can use the following guideline: If your application accesses non-static tables, and there is no risk of deadlock, have your program execute the following statements when it starts up (immediately after the first CONNECT or DATABASE statement):

```
SET LOCK MODE TO WAIT
SET ISOLATION TO REPEATABLE READ
```

Ignore the return codes from both statements. Proceed as if no other programs exist. If no performance problems arise, you do not need to read this section again.

Hold cursors

When transaction logging is used, HCL Informix® guarantees that anything done within a transaction can be rolled back at the end of it. To handle transactions reliably, the database server normally applies the following rules:

- When a transaction ends, all cursors are closed.
- When a transaction ends, all locks are released.

The rules that are used to handle transactions reliably are normal with most database systems that support transactions, and they do not cause any trouble for most applications. However, circumstances exist in which using standard transactions with cursors is not possible. For example, the following code works fine without transactions. However, when transactions are added, closing the cursor conflicts with using two cursors simultaneously.

```
EXEC SQL DECLARE master CURSOR FOR
EXEC SQL DECLARE detail CURSOR FOR  FOR UPDATE
EXEC SQL OPEN master;
while(SQLCODE == 0)
{
   EXEC SQL FETCH master INTO
   if(SQLCODE == 0)
   {
      EXEC SQL BEGIN WORK;
      EXEC SQL OPEN detail USING
      EXEC SQL FETCH detail
      EXEC SQL UPDATE  WHERE CURRENT OF detail
      EXEC SQL COMMIT WORK;
   }
}
EXEC SQL CLOSE master;
```

In this design, one cursor is used to scan a table. Selected records are used as the basis for updating a different table. The problem is that when each update is treated as a separate transaction (as the pseudocode in the previous example shows), the COMMIT WORK statement following the UPDATE closes all cursors, including the master cursor.

The simplest alternative is to move the COMMIT WORK and BEGIN WORK statements to be the last and first statements, respectively, so that the entire scan over the master table is one large transaction. Treating the scan of the master table as one large transaction is sometimes possible, but it can become impractical if many rows need to be updated. The number of locks can be too large, and they are held for the duration of the program.

A solution that HCL® Informix® database servers support is to add the keywords WITH HOLD to the declaration of the master cursor. Such a cursor is referred to as a *hold cursor* and is not closed at the end of a transaction. The database server still closes all other cursors, and it still releases all locks, but the hold cursor remains open until it is explicitly closed.

Before you attempt to use a hold cursor, you must be sure that you understand the locking mechanism described here, and you must also understand the programs that are running concurrently. Whenever COMMIT WORK is executed, all locks are released, including any locks placed on rows fetched through the hold cursor.

The removal of locks has little importance if the cursor is used as intended, for a single forward scan over a table. However, you can specify WITH HOLD for any cursor, including update cursors and scroll cursors. Before you do this, you must understand the implications of the fact that all locks (including locks on entire tables) are released at the end of a transaction.

The SQL statement cache

The SQL statement cache is a feature that lets you store in a buffer identical SQL statements that are executed repeatedly so the statements can be reused among different user sessions without the need for per-session memory allocation. Statement caching can dramatically improve performance for applications that contain a large number of prepared statements. However, performance improvements are less dramatic when statement caching is used to cache statements that are prepared once and executed many times.

Use SQL to turn on or turn off statement caching for an individual database session when statement caching is enabled for the database server. The following statement shows how to use SQL to turn on caching for the current database session:

```
SET STATEMENT CACHE ON
```

The following statement shows how to use SQL to turn off caching for the current database session:

```
SET STATEMENT CACHE OFF
```

If you attempt to turn on or turn off statement caching when caching is disabled, the database server returns an error.

For information about syntax for the SET STATEMENT CACHE statement, see the *HCL® Informix® Guide to SQL: Syntax*. For information about the STMT_CACHE and STMT_CACHE_SIZE configuration parameters, see the *HCL® Informix® Administrator's Reference* and your . For information about the **STMT_CACHE** environment variable, see the *HCL® Informix® Guide to SQL: Reference*.

Summary

Whenever multiple programs have access to a database concurrently (and when at least one of them can modify data), all programs must allow for the possibility that another program can change the data even as they read it. The database server provides a mechanism of locks and isolation levels that usually allow programs to run as if they were alone with the data.

The SET STATEMENT CACHE statement allows you to store in a buffer identical SQL statements that are used repeatedly. When statement caching is turned on, the database server stores the identical statements so they can be reused among different user sessions without the need for per-session memory allocation.

Create and use SPL routines

This section describes how to create and use SPL routines. An SPL routine is a user-defined routine written in HCL® Informix® Stored Procedure Language (SPL). HCL® Informix® SPL is an extension to SQL that provides flow control, such as looping and branching. Anyone who has the Resource privilege on a database can create an SPL routine.

Routines written in SQL are parsed, optimized as far as possible, and then stored in the system catalog tables in executable format. An SPL routine might be a good choice for SQL-intensive tasks. SPL routines can execute routines written in C or other external languages, and external routines can execute SPL routines.

You can use SPL routines to perform any task that you can perform in SQL and to expand what you can accomplish with SQL alone. Because SPL is a language native to the database, and because SPL routines are parsed and optimized when they are created rather than at runtime, SPL routines can improve performance for some tasks. SPL routines can also reduce traffic between a client application and the database server and reduce program complexity.

The syntax for each SPL statement is described in the *HCL® Informix® Guide to SQL: Syntax*. Examples accompany the syntax for each statement.

Introduction to SPL routines

An SPL *routine* is a generic term that includes SPL *procedures* and SPL *functions*. An SPL procedure is a routine written in SPL and SQL that does not return a value. An SPL function is a routine written in SPL and SQL that returns a single value, a value with a complex data type, or multiple values. Generally, a routine written in SPL that returns a value is an SPL function.

You use SQL and SPL statements to write an SPL routine. SPL statements can be used only inside the CREATE PROCEDURE, CREATE PROCEDURE FROM, CREATE FUNCTION, and CREATE FUNCTION FROM statements. All these statements are available with SQL APIs such as . The CREATE PROCEDURE and CREATE FUNCTION statements are available with .

To list all SPL routines in a database, run this command, which creates and displays the schema for a database:

```
dbschema -d database_name -f all
```

What you can do with SPL routines

You can accomplish a wide range of objectives with SPL routines, including improving database performance, simplifying writing applications, and limiting or monitoring access to data.

Because an SPL routine is stored in an executable format, you can use it to execute frequently repeated tasks to improve performance. When you execute an SPL routine rather than straight SQL code, you can bypass repeated parsing, validity checking, and query optimization.

You can use an SPL routine in a data-manipulation SQL statement to supply values to that statement. For example, you can use a routine to perform the following actions:

- Supply values to be inserted into a table
- Supply a value that makes up part of a condition clause in a SELECT, DELETE, or UPDATE statement

These actions are two possible uses of a routine in a data-manipulation statement, but others exist. In fact, any expression in a data-manipulation SQL statement can consist of a routine call.

You can also issue SQL statements in an SPL routine to hide those SQL statements from a database user. Rather than having all users learn how to use SQL, one experienced SQL user can write an SPL routine to encapsulate an SQL activity and let others know that the routine is stored in the database so that they can execute it.

You can write an SPL routine to be run with the DBA privilege by a user who does not have the DBA privilege. This feature allows you to limit and control access to data in the database. Alternatively, an SPL routine can monitor the users who access certain tables or data. For more information about how to use SPL routines to control access to data, see the *IBM® Informix® Database Design and Implementation Guide*.

You also can write SPL routines that use Dynamic SQL. For an overview with detailed examples of how to create and use prepared objects and Dynamic SQL in SPL routines, see this IBM® developerWorks® article: Dynamic SQL support in Informix® Dynamic Server Stored Procedure Language.

SPL routines format

An SPL routine consists of a beginning statement, a statement block, and an ending statement. Within the statement block, you can use SQL or SPL statements.

The CREATE PROCEDURE or CREATE FUNCTION statement

You must first decide if the routine that you are creating returns values or not. If the routine does not return a value, use the CREATE PROCEDURE statement to create an SPL procedure. If the routine returns a value, use the CREATE FUNCTION statement to create an SPL function.

To create an SPL routine, use one CREATE PROCEDURE or CREATE FUNCTION statement to write the body of the routine and register it.

Begin and end the routine

To create an SPL routine that does not return values, start with the CREATE PROCEDURE statement and end with the END PROCEDURE keyword. The following figure shows how to begin and end an SPL procedure.

Figure 383. Begin and end an SPL procedure.

```
CREATE PROCEDURE new_price( per_cent REAL )
. . .
END PROCEDURE;
```

For more information about naming conventions, see the Identifier segment in the *HCL® Informix® Guide to SQL: Syntax*.

To create an SPL function that returns one or more values, start with the CREATE FUNCTION statement and end with the END FUNCTION keyword. The following figure shows how to begin and end an SPL function.

Figure 384. Begin and end an SPL function.

```
CREATE FUNCTION discount_price( per_cent REAL)
   RETURNING MONEY;
. . .
END FUNCTION;
```

In SPL routines, the END PROCEDURE or END FUNCTION keywords are required.

Important: For compatibility with earlier HCL Informix® products, you can use CREATE PROCEDURE with a RETURNING clause to create a user-defined routine that returns a value. Your code will be easier to read and to maintain, however, it you use CREATE PROCEDURE for SPL routines that do not return values (SPL procedures) and CREATE FUNCTION for SPL routines that return one or more values (SPL functions).

Specify a routine name

You specify a name for the SPL routine immediately following the CREATE PROCEDURE or CREATE FUNCTION statement and before the parameter list, as the figure shows.

Figure 385. Specify a name for the SPL routine.

```
CREATE PROCEDURE add_price (arg INT )
```

HCL Informix® allows you to create more than one SPL routine with the same name but with different parameters. This feature is known as *routine overloading*. For example, you might create each of the following SPL routines in your database:

```
CREATE PROCEDURE multiply (a INT, b FLOAT)
CREATE PROCEDURE multiply (a INT, b SMALLINT)
CREATE PROCEDURE multiply (a REAL, b REAL)
```

If you call a routine with the name **multiply()**, the database server evaluates the name of the routine and its arguments to determine which routine to execute.

Routine resolution is the process in which the database server searches for a routine signature that it can use, given the name of the routine and a list of arguments. Every routine has a *signature* that uniquely identifies the routine based on the following information:

- The type of routine (procedure or function)
- The routine name
- The number of parameters
- The data types of the parameters
- The order of the parameters

The routine signature is used in a CREATE, DROP, or EXECUTE statement if you enter the full parameter list of the routine. For example, each statement in the following figure uses a routine signature.

Figure 386. Routine signatures.

```
CREATE FUNCTION multiply(a INT, b INT);

DROP PROCEDURE end_of_list(n SET, row_id INT);

EXECUTE FUNCTION compare_point(m point, n point);
```

Add a specific name

Because HCL Informix® supports routine overloading, an SPL routine might not be uniquely identified by its name alone. However, a routine can be uniquely identified by a *specific name*. A specific name is a unique identifier that you define in the CREATE PROCEDURE or CREATE FUNCTION statement, in addition to the routine name. A specific name is defined with the SPECIFIC keyword and is unique in the database. Two routines in the same database cannot have the same specific name, even if they have different owners.

A specific name can be up to 128 bytes long. The following figure shows how to define the specific name **calc1** in a CREATE FUNCTION statement that creates the calculate() function.

Figure 387. Define the specific name.

```
CREATE FUNCTION calculate(a INT, b INT, c INT)
   RETURNING INT
   SPECIFIC calc1;
. . .
END FUNCTION;
```

Because the owner **bsmith** has given the SPL function the specific name **calc1**, no other user can define a routine-SPL or external-with the specific name **calc1**. Now you can refer to the routine as **bsmith.calculate** or with the SPECIFIC keyword **calc1** in any statement that requires the SPECIFIC keyword.

Add a parameter list

When you create an SPL routine, you can define a parameter list so that the routine accepts one or more arguments when it is invoked. The parameter list is optional.

A parameter to an SPL routine must have a name and can be defined with a default value. The following are the categories of data types that a parameter can specify:

- Built-in data types
- Opaque data types
- Distinct data types
- Row types
- Collection types
- Smart large objects (CLOB and BLOB)

The parameter list cannot specify any of the following data types directly:

- SERIAL
- SERIAL8
- BIGSERIAL
- TEXT
- BYTE

For the serial data types, however, a routine can return numerically equivalent values that are cast to a corresponding integer type (INT, INT8, or BIGINT). Similarly, for a routine to support the simple large object data types, the parameter list can include the REFERENCES keyword to return a descriptor that points to the storage location of the TEXT or BYTE object.

The following figure shows examples of parameter lists.

Figure 388. Examples of different parameter lists.

```
CREATE PROCEDURE raise_price(per_cent INT);

CREATE FUNCTION  raise_price(per_cent INT DEFAULT 5);

CREATE PROCEDURE update_emp(n employee_t);
CREATE FUNCTION  update_nums( list1 LIST(ROW (a VARCHAR(10),
                                              b VARCHAR(10),
                                              c INT) NOT NULL ));
```

When you define a parameter, you accomplish two tasks at once:

- You request that the user supply a value when the routine is executed.
- You implicitly define a variable (with the same name as the parameter name) that you can use as a local variable in the body of the routine.

If you define a parameter with a default value, the user can execute the SPL routine with or without the corresponding argument. If the user executes the SPL routine without the argument, the database server assigns the parameter the default value as an argument.

When you invoke an SPL routine, you can give an argument a NULL value. SPL routines handle NULL values by default. However, you cannot give an argument a NULL value if the argument is a collection element.

Simple large objects as parameters

Although you cannot define a parameter with a simple large object (a large object that contains TEXT or BYTE data types), you can use the REFERENCES keyword to define a parameter that points to a simple large object, as the following figure shows.

Figure 389. Use of the REFERENCES keyword.

```
CREATE PROCEDURE proc1(lo_text REFERENCES TEXT)

CREATE FUNCTION proc2(lo_byte REFERENCES BYTE DEFAULT NULL)
```

The REFERENCES keyword means that the SPL routine is passed a descriptor that contains a pointer to the simple large object, not the object itself.

Undefined arguments

When you invoke an SPL routine, you can specify all, some, or none of the defined arguments. If you do not specify an argument, and if its corresponding parameter does not have a default value, the argument, which is used as a variable within the SPL routine, is given a status of *undefined*.

Undefined is a special status used for SPL variables that have no value. The SPL routine executes without error, as long as you do not attempt to use the variable that has the status undefined in the body of the routine.

The undefined status is not the same as a NULL value. (The NULL value means that the value is not known, or does not exist, or is not applicable.)

Add a return clause

If you use CREATE FUNCTION to create an SPL routine, you must specify a return clause that returns one or more values.

Tip: If you use the CREATE PROCEDURE statement to create an SPL routine, you have the option of specifying a return clause. Your code will be easier to read and to maintain, however, it you instead use the CREATE FUNCTION statement to create a routine that returns values.

To specify a return clause, use the RETURNING or RETURNS keyword with a list of data types the routine will return. The data types can be any SQL data types except SERIAL, SERIAL8, TEXT, or BYTE.

The return clause in the following figure specifies that the SPL routine will return an INT value and a REAL value.

Figure 390. Specify the return clause.

```
FUNCTION find_group(id INT)
   RETURNING INT, REAL;
. . .
END FUNCTION;
```

After you specify a return clause, you must also specify a RETURN statement in the body of the routine that explicitly returns the values to the calling routine. For more information on writing the RETURN statement, see Return values from an SPL function on page 1782.

To specify that the function should return a simple large object (a TEXT or BYTE value), you must use the REFERENCES clause, as in the following figure, because an SPL routine returns only a pointer to the object, not the object itself.

Figure 391. Use of the REFERENCES clause.

```
CREATE FUNCTION find_obj(id INT)
   RETURNING REFERENCES BYTE;
```

Add display labels

You can use CREATE FUNCTION to create a routine that specifies names for the display labels for the values returned. If you do not specify names for the display labels, the labels will display as **expression**.

In addition, although using CREATE FUNCTION for routines that return values is recommended, you can use CREATE PROCEDURE to create a routine that returns values and specifies display labels for the values returned.

If you choose to specify a display label for one return value, you must specify a display label for every return value. In addition, each return value must have a unique display label.

To add display labels, you must specify a return clause, use the RETURNING keyword. The return clause in the following figure specifies that the routine will return an INT value with a `serial_num` display label, a CHAR value with a `name` display label, and an INT value with a `points` display label. You could use either CREATE FUNCTION or CREATE PROCEDURE in the following figure.

Figure 392. Specify the return clause.

```
CREATE FUNCTION p(inval INT DEFAULT 0)
   RETURNING INT AS serial_num, CHAR (10) AS name, INT AS points;
   RETURN (inval + 1002), "Newton", 100;
END FUNCTION;
```

The returned values and their display labels are shown in the following figure.

Figure 393. Returned values and their display labels.

```
serial_num    name      points

1002          Newton    100
```

Tip: Because you can specify display labels for return values directly in a SELECT statement, when a SPL routine is used in a SELECT statement, the labels will display as **expression**. For more information on specifying display labels for return values in a SELECT statement, see Compose SELECT statements on page 1532.

Specify whether the SPL function is variant

When you create an SPL function, the function is variant by default. A function is variant if it returns different results when it is invoked with the same arguments or if it modifies a database or variable state. For example, a function that returns the current date or time is a variant function.

Although SPL functions are variant by default, if you specify WITH NOT VARIANT when you create a function, the function cannot contain any SQL statements. You can define a functional index only on a nonvariant function.

Add a modifier

When you write SPL functions, you can use the WITH clause to add a modifier to the CREATE FUNCTION statement. In the WITH clause, you can specify the COMMUTATOR or NEGATOR functions. The other modifiers are for external routines.

Restriction: You can use the COMMUTATOR or NEGATOR modifiers with SPL functions only. You cannot use any modifiers with SPL procedures.

The COMMUTATOR modifier

The COMMUTATOR modifier allows you to specify an SPL function that is the *commutator function* of the SPL function you are creating. A commutator function accepts the same arguments as the SPL function you are creating, but in opposite order, and returns the same value. The commutator function might be more cost effective for the SQL optimizer to execute.

For example, the functions lessthan(a,b), which returns TRUE if **a** is less than **b**, and greaterthan(b,a), which returns TRUE if **b** is greater than or equal to **a**, are commutator functions. The following figure uses the WITH clause to define a commutator function.

Figure 394. Define a commutator function.

```
CREATE FUNCTION lessthan( a dtype1, b dtype2 )
   RETURNING BOOLEAN
   WITH ( COMMUTATOR = greaterthan );
. . .
END FUNCTION;
```

The optimizer might use greaterthan(b,a) if it is less expensive to execute than lessthan(a,b). To specify a commutator function, you must own both the commutator function and the SPL function you are writing. You must also grant the user of your SPL function the Execute privilege on both functions.

For a detailed description of granting privileges, see the description of the GRANT statement in the *HCL® Informix® Guide to SQL: Syntax*.

The NEGATOR modifier

The NEGATOR modifier is available for Boolean functions. Two Boolean functions are *negator functions* if they take the same arguments, in the same order, and return complementary Boolean values.

For example, the functions equal(a,b), which returns TRUE if **a** is equal to **b**, and notequal(a,b), which returns FALSE if **a** is equal to **b,** are negator functions. The optimizer might choose to execute the negator function you specify if it is less expensive than the original function.

Tthe following figure shows how to use the WITH clause of a CREATE FUNCTION statement to specify a negator function.

Figure 395. Specify a negator function.

```
CREATE FUNCTION equal( a dtype1, b dtype2 )
   RETURNING BOOLEAN
   WITH ( NEGATOR = notequal );
. . .
END FUNCTION;
```

Tip: By default, any SPL routine can handle NULL values that are passed to it in the argument list. In other words, the HANDLESNULLS modifier is set to YES for SPL routines, and you cannot change its value.

For more information on the COMMUTATOR and NEGATOR modifiers, see the Routine Modifier segment in the *HCL® Informix® Guide to SQL: Syntax*.

Specify a DOCUMENT clause

The DOCUMENT and WITH LISTING IN clauses follow END PROCEDURE or END FUNCTION statements.

The DOCUMENT clause lets you add comments to your SPL routine that another routine can select from the system catalog tables, if needed. The DOCUMENT clause in the following figure contains a usage statement that shows a user how to run the SPL routine.

Figure 396. Usage statement that shows a user how to run the SPL routine.

```
CREATE FUNCTION raise_prices(per_cent INT)
. . .
END FUNCTION
   DOCUMENT "USAGE: EXECUTE FUNCTION raise_prices (xxx)",
            "xxx = percentage from 1 - 100";
```

Remember to place single or double quotation marks around the literal clause. If the literal clause extends past one line, place quotation marks around each line.

Specify a listing file

The WITH LISTING IN option allows you to direct any compile-time warnings that might occur to a file.

The following figure shows how to log the compile-time warnings in `/tmp/warn_file` when you work on UNIX™.

Figure 397. Log the compile-time warnings on UNIX™.

```
CREATE FUNCTION raise_prices(per_cent INT)
. . .
END FUNCTION
   WITH LISTING IN '/tmp/warn_file'
```

The following figure shows how to log the compile-time warnings in `\tmp\listfile` when you work on Windows™.

Figure 398. Log the compile-time warnings on Windows™.

```
CREATE FUNCTION raise_prices(per_cent INT)
. . .
END FUNCTION
   WITH LISTING IN 'C:\tmp\listfile'
```

Always remember to place single or double quotation marks around the file name or path name.

Add comments

You can add a comment to any line of an SPL routine, even a blank line.

To add a comment, use any of the following comment notation styles:

- Place a double hyphen (`--`) at the left of the comment.
- Enclose the comment text between a pair of braces (`{` .. .`}`).
- Delimit the comment between C-style "slash and asterisk" comment indicators (`/*` .. .`*/`).

To add a multiple-line comment, take one of the following actions:

- Place a double hyphen before each line of the comment
- Enclose the entire comment within the pair of braces.
- Place `/*` at the left of the first line of the comment, and place `*/` at the end of the last line of the comment.

Braces as comment indicators are HCL® Informix® an extension to the ANSI/ISO standard for the SQL language. All three comment styles are also valid in SPL routines.

If you use braces or C-style comment indicators to delimit the text of a comment, the opening indicator must be in the same style as the closing indicator.

All the examples in the following figure are valid comments.

Figure 399. Valid comment examples.

```
SELECT * FROM customer -- Selects all columns and rows

SELECT * FROM customer
   -- Selects all columns and rows
   -- from the customer table

SELECT * FROM customer
   { Selects all columns and rows
     from the customer table }

SELECT * FROM customer
   /* Selects all columns and rows
     from the customer table */
```

Important: Braces ({ }) can be used to delimit comments and also to delimit the list of elements in a collection. To ensure that the parser correctly recognizes the end of a comment or list of elements in a collection, use the double hyphen (--) for comments in an SPL routine that handles collection data types.

Example of a complete routine

The following CREATE FUNCTION statement creates a routine that reads a customer address:

```
CREATE FUNCTION read_address       (lastname CHAR(15)) -- one argument
   RETURNING CHAR(15), CHAR(15), CHAR(20), CHAR(15),CHAR(2)
      CHAR(5); -- 6 items

   DEFINE p_lname,p_fname, p_city CHAR(15);
      --define each routine variable
```

```
   DEFINE p_add CHAR(20);
   DEFINE p_state CHAR(2);
   DEFINE p_zip CHAR(5);

   SELECT fname, address1, city, state, zipcode
      INTO p_fname, p_add, p_city, p_state, p_zip
      FROM customer
      WHERE lname = lastname;

   RETURN p_fname, lastname, p_add, p_city, p_state, p_zip;
      --6 items
END FUNCTION;

DOCUMENT 'This routine takes the last name of a customer as',
   --brief description
   'its only argument. It returns the full name and address',
   'of the customer.'

WITH LISTING IN 'pathname' -- modify this pathname according
-- to the conventions that your operating system requires

-- compile-time warnings go here
; -- end of the routine read_address
```

Create an SPL routine in a program

To use an SQL API to create an SPL routine, put the text of the CREATE PROCEDURE or CREATE FUNCTION statement in a file. Use the CREATE PROCEDURE FROM or CREATE FUNCTION FROM statement and refer to that file to compile the routine. For example, to create a routine to read a customer name, you can use a statement such as the one in the previous example and store it in a file. If the file is named `read_add_source`, the following statement compiles the **read_address** routine:

```
CREATE PROCEDURE FROM 'read_add_source';
```

The following example shows how the previous SQL statement looks in the program:

```
/* This program creates whatever routine is in *
 * the file 'read_add_source'.
 */
#include <stdio.h>
EXEC SQL include sqlca;
EXEC SQL include sqlda;
EXEC SQL include datetime;
/* Program to create a routine from the pwd */

main()
{
EXEC SQL database play;
EXEC SQL create procedure from 'read_add_source';
}
```

Dropping a routine in a local or remote database

After you create an SPL routine, you cannot change the body of the routine. Instead, you need to drop the routine and re-create it. Before you drop the routine, however, make sure that you have a copy of its text somewhere outside the database.

In general, use DROP PROCEDURE with an SPL procedure name and DROP FUNCTION with an SPL function name, as the following figure shows.

Figure 400. DROP PROCEDURE and DROP FUNCTION.

```
DROP PROCEDURE raise_prices;
DROP FUNCTION read_address;
```

Tip: You can also use DROP PROCEDURE with a function name to drop an SPL function. However, it is recommended that you use DROP PROCEDURE only with procedure names and DROP FUNCTION only with function names.

If the database has other routines of the same name (overloaded routines), you cannot drop the SPL routine by its routine name alone. To drop a routine that has been overloaded, you must specify either its signature or its specific name. The following figure shows two ways that you might drop a routine that is overloaded.

Figure 401. Drop a routine that is overloaded.

```
DROP FUNCTION calculate(INT, INT, INT);
   -- this is a signature

DROP SPECIFIC FUNCTION calc1;
   -- this is a specific name
```

If you do not know the type of a routine (function or procedure), you can use the DROP ROUTINE statement to drop it. DROP ROUTINE works with either functions or procedures. DROP ROUTINE also has a SPECIFIC keyword, as the following figure shows.

Figure 402. The DROP ROUTINE statement.

```
DROP ROUTINE calculate;
DROP SPECIFIC ROUTINE calc1;
```

Before you drop an SPL routine stored on a remote database server, be aware of the following restriction. You can drop an SPL routine with a fully qualified routine name in the form `database@dbservername:owner.routinename` only if the routine name alone, without its arguments, is enough to identify the routine.

Restrictions on data types in distributed operations

SPL routines that access tables in databases of non-local database servers, or that are invoked as UDRs of a database of another database server, can only have non-opaque built-in data types as their arguments or returned values.

If the tables or the UDR resides on another database of the same Informix® instance, however, the arguments and returned values of routines written in SPL (or in external languages that Informix® supports) can be the built-in opaque data types BLOB, BOOLEAN, CLOB, and LVARCHAR. They can also be UDTs or DISTINCT data types if the following conditions are true:

- The remote database has the same server as the current database.
- The UDT arguments are explicitly cast to a built-in data type.
- The DISTINCT types are based on built-in types and are explicitly cast to built-in types.
- The SPL routine and all the casts are defined in all participating databases.

Define and use variables

Any variable that you use in an SPL routine, other than a variable that is implicitly defined in the parameter list of the routine, must be defined in the body of the routine.

The value of a variable is held in memory; the variable is not a database object. Therefore, rolling back a transaction does not restore the values of SPL variables.

To define a variable in an SPL routine, use the DEFINE statement. DEFINE is not an executable statement. DEFINE must appear after the CREATE PROCEDURE statement and before any other statements. The examples in the following figure are all legal variable definitions.

Figure 403. Variable definitions.

```
DEFINE a INT;
DEFINE person person_t;
DEFINE GLOBAL gl_out INT DEFAULT 13;
```

For more information on DEFINE, see the description in the *HCL® Informix® Guide to SQL: Syntax*.

An SPL variable has a name and a data type. The variable name must be a valid identifier, as described in the Identifier segment in the *HCL® Informix® Guide to SQL: Syntax*.

Declare local variables

You can define a variable to be either *local* or *global* in scope. This section describes local variables. In an SPL routine, local variables:

- Are valid only for the duration of the SPL routine
- Are reset to their initial values or to a value the user passes to the routine, each time the routine is executed
- Cannot have default values

You can define a local variable on any of the following data types:

- Built-in data types (except SERIAL, SERIAL8, BIGSERIAL, TEXT, or BYTE)
- Any extended data type (row type, opaque, distinct, or collection type) that is defined in the database prior to execution of the SPL routine

The scope of a local variable is the statement block in which it is declared. You can use the same variable name outside the statement block with a different definition.

For more information on defining global variables, see Declare global variables on page 1772.

Scope of local variables

A local variable is valid within the statement block in which it is defined and within any nested statement blocks, unless you redefine the variable within the statement block.

In the beginning of the SPL procedure in the following figure, the integer variables **x**, **y**, and **z** are defined and initialized.

Figure 404. Define and initialize variables.

```
CREATE PROCEDURE scope()
   DEFINE x,y,z INT;
   LET x = 5;
   LET y = 10;
   LET z = x + y; --z is 15
   BEGIN
      DEFINE x, q INT;
      DEFINE z CHAR(5);
      LET x = 100;
      LET q = x + y;   -- q = 110
      LET z = 'silly'; -- z receives a character value
   END
   LET y = x; -- y is now 5
   LET x = z; -- z is now 15, not 'silly'
END PROCEDURE;
```

The BEGIN and END statements mark a nested statement block in which the integer variables **x** and **q** are defined as well as the CHAR variable **z**. Within the nested block, the redefined variable **x** masks the original variable **x**. After the END statement, which marks the end of the nested block, the original value of **x** is accessible again.

Declare variables of built-in data types

A variable that you declare as a built-in SQL data type can hold a value retrieved from a column of that built-in type. You can declare an SPL variable as most built-in types, except BIGSERIAL, SERIAL, and SERIAL8, as the following figure illustrates.

Figure 405. Built-in type variables.

```
DEFINE x INT;
DEFINE y INT8;
DEFINE name CHAR(15);
DEFINE this_day DATETIME YEAR TO DAY;
```

You can declare SPL variables of appropriate integer data types (such as BIGINT, INT, or INT8) to store the values of serial columns or of sequence objects.

Declare variables for smart large objects

A variable for a BLOB or CLOB object (or a data type that contains a smart large object) does not contain the object itself but rather a pointer to the object. The following figure shows how to define a variable for BLOB and CLOB objects.

Figure 406. Variables for BLOB or CLOB objects.

```
DEFINE a_blob BLOB;
DEFINE b_clob CLOB;
```

Declare variables for simple large objects

A variable for a simple large object (a TEXT or BYTE object) does not contain the object itself but rather a pointer to the object. When you define a variable on the TEXT or BYTE data type, you must use the keyword REFERENCES before the data type, as the following figure shows.

Figure 407. Use the REFERENCES keyword before the data type.

```
DEFINE t REFERENCES TEXT;
DEFINE b REFERENCES BYTE;
```

Declare collection variables

In order to hold a collection fetched from the database, a variable must be of type SET, MULTISET, or LIST.

Important: A collection variable must be defined as a local variable. You cannot define a collection variable as a global variable.

A variable of SET, MULTISET, or LIST type is a collection variable that holds a collection of the type named in the DEFINE statement. The following figure shows how to define typed collection variables.

Figure 408. Define typed collection variables.

```
DEFINE a SET ( INT NOT NULL );

DEFINE b MULTISET ( ROW (  b1 INT,
                           b2 CHAR(50),
                        )  NOT NULL );

DEFINE c LIST ( SET (DECIMAL NOT NULL) NOT NULL);
```

You must always define the elements of a collection variable as NOT NULL. In this example, the variable **a** is defined to hold a SET of non-NULL integers; the variable **b** holds a MULTISET of non-NULL row types; and the variable **c** holds a LIST of non-NULL sets of non-NULL decimal values.

In a variable definition, you can nest complex types in any combination or depth to match the data types stored in your database.

You cannot assign a collection variable of one type to a collection variable of another type. For example, if you define a collection variable as a SET, you cannot assign another collection variable of MULTISET or LIST type to it.

Declare row-type variables

Row-type variables hold data from named or unnamed row types. You can define a *named row variable* or an *unnamed row variable*. Suppose you define the named row types that the following figure shows.

Figure 409. Named and unnamed row variables.

```
CREATE ROW TYPE zip_t
(
   z_code      CHAR(5),
   z_suffix    CHAR(4)
);

CREATE ROW TYPE address_t
(
   street      VARCHAR(20),
   city        VARCHAR(20),
   state       CHAR(2),
   zip         zip_t
);

CREATE ROW TYPE employee_t
(
   name        VARCHAR(30),
   address     address_t
   salary      INTEGER
);

CREATE TABLE employee OF TYPE employee_t;
```

If you define a variable with the name of a named row type, the variable can only hold data of that row type. In the following figure, the **person** variable can only hold data of **employee_t** type.

Figure 410. Defining the person variable.

```
DEFINE person employee_t;
```

To define a variable that holds data stored in an unnamed row type, use the ROW keyword followed by the fields of the row type, as the following figure shows.

Figure 411. Use the ROW keyword followed by the fields of the row type.

```
DEFINE manager ROW (name        VARCHAR(30),
                    department  VARCHAR(30),
                    salary      INTEGER );
```

Because unnamed row types are type-checked for structural equivalence only, a variable defined with an unnamed row type can hold data from any unnamed row type that has the same number of fields and the same type definitions. Therefore, the variable **manager** can hold data from any of the row types in the following figure.

Figure 412. Unnamed row types.

```
ROW ( name         VARCHAR(30),
      department   VARCHAR(30),
      salary       INTEGER );

ROW ( french      VARCHAR(30),
      spanish     VARCHAR(30),
      number      INTEGER );

ROW ( title       VARCHAR(30),
      musician    VARCHAR(30),
      price       INTEGER );
```

Important: Before you can use a row type variable, you must initialize the row variable with a LET statement or SELECTINTO statement.

Declare opaque- and distinct-type variables

Opaque-type variables hold data retrieved from opaque data types. *Distinct-type variables* hold data retrieved from distinct data types. If you define a variable with an opaque data type or a distinct data type, the variable can only hold data of that type.

If you define an opaque data type named **point** and a distinct data type named **centerpoint**, you can define SPL variables to hold data from the two types, as the following figure shows.

Figure 413. Defining SPL variables to hold opaque and distinct data types.

```
DEFINE a point;
DEFINE b centerpoint;
```

The variable **a** can only hold data of type **point**, and **b** can only hold data of type **centerpoint**.

Declare variables for column data with the LIKE clause

If you use the LIKE clause, the database server defines a variable to have the same data type as a column in a table or view.

If the column contains a collection, row type, or nested complex type, the variable has the complex or nested complex type defined in the column.

In the following figure, the variable **loc1** defines the data type for the **locations** column in the **image** table.

Figure 414. Define the loc1 data type for the locations column in the image table.

```
DEFINE loc1 LIKE image.locations;
```

Declare PROCEDURE type variables

In an SPL routine, you can define a variable of type PROCEDURE and assign the variable the name of an existing SPL routine or external routine. Defining a variable of PROCEDURE type indicates that the variable is a call to a user-defined routine, not a built-in routine of the same name.

For example, the statement in the following figure defines **length** as an SPL procedure or SPL function, not as the built-in LENGTH function.

Figure 415. Define length as an SPL procedure.

```
DEFINE length PROCEDURE;
LET x = length( a,b,c );
```

This definition disables the built-in LENGTH function within the scope of the statement block. You would use such a definition if you had already created an SPL or external routine with the name LENGTH.

Because HCL Informix® supports routine overloading, you can define more than one SPL routine or external routine with the same name. If you call any routine from an SPL routine, Informix® determines which routine to use, based on the arguments specified and the routine determination rules. For information about routine overloading and routine determination, see *HCL® Informix® User-Defined Routines and Data Types Developer's Guide*.

Tip: If you create an SPL routine with the same name as an aggregate function (SUM, MAX, MIN, AVG, COUNT) or with the name **extend**, you must qualify the routine name with an owner name.

Subscripts with variables

You can use subscripts with variables of CHAR, VARCHAR, NCHAR, NVARCHAR, BYTE, or TEXT data type. The subscripts indicate the starting and ending character positions that you want to use within the variable.

Subscripts must always be constants. You cannot use variables as subscripts. The following figure illustrates how to use a subscript with a CHAR(15) variable.

Figure 416. A subscript with a CHAR(15) variable.

```
DEFINE name CHAR(15);
LET name[4,7] = 'Ream';
SELECT fname[1,3] INTO name[1,3] FROM customer
   WHERE lname = 'Ream';
```

In this example, the customer's last name is placed between positions 4 and 7 of **name**. The first three characters of the customer's first name is retrieved into positions 1 through 3 of **name**. The part of the variable that is delimited by the two subscripts is referred to as a *substring*.

Variable and keyword ambiguity

If you declare a variable whose name is an SQL keyword, ambiguities can occur. The following rules for identifiers help you avoid ambiguities for SPL variables, SPL routine names, and built-in function names:

- Defined variables take the highest precedence.
- Routines defined with the PROCEDURE keyword in a DEFINE statement take precedence over SQL functions.
- SQL functions take precedence over SPL routines that exist but are not identified with the PROCEDURE keyword in a DEFINE statement.

In general, avoid using an ANSI-reserved word for the name of the variable. For example, you cannot define a variable with the name **count** or **max** because they are the names of aggregate functions. For a list of the reserved keywords that you should avoid using as variable names, see the Identifier segment in the *HCL® Informix® Guide to SQL: Syntax*.

For information about ambiguities between SPL routine names and SQL function names, see the *HCL® Informix® Guide to SQL: Syntax*.

Variables and column names

If you use the same identifier for an SPL variable that you use for a column name, the database server assumes that each instance of the identifier is a variable. Qualify the column name with the table name, using dot notation, in order to use the identifier as a column name.

In the SELECT statement in the following figure, **customer.lname** is a column name and **lname** is a variable name.

Figure 417. Column names and variable names in a SELECT statement.

```
CREATE PROCEDURE table_test()

   DEFINE lname CHAR(15);
   LET lname = 'Miller';

   SELECT customer.lname INTO lname FROM customer
      WHERE customer_num = 502;
. . .
END PROCEDURE;
```

Variables and SQL functions

If you use the same identifier for an SPL variable as for an SQL function, the database server assumes that an instance of the identifier is a variable and disallows the use of the SQL function. You cannot use the SQL function within the block of code in which the variable is defined. The example in the following figure shows a block within an SPL procedure in which the variable called **user** is defined. This definition disallows the use of the USER function in the BEGIN END block.

Figure 418. A procedure that disallows the use of the USER function in the BEGIN END block.

```
CREATE PROCEDURE user_test()
   DEFINE name CHAR(10);
   DEFINE name2 CHAR(10);
   LET name = user; -- the SQL function

   BEGIN
      DEFINE user CHAR(15); -- disables user function
      LET user = 'Miller';
      LET name = user; -- assigns 'Miller' to variable name
   END
   . . .
   LET name2 = user; -- SQL function again
```

Declare global variables

A *global variable* has its value stored in memory and is available to other SPL routines, run by the same user session, on the same database. A global variable has the following characteristics:

- It requires a default value.
- It can be used in any SPL routine, although it must be defined in each routine in which it is used.
- It carries its value from one SPL routine to another until the session ends.

Restriction: You cannot define a collection variable as a global variable.

The following figure shows two SPL functions that share a global variable.

Figure 419. Two SPL functions that share a global variable.

```
CREATE FUNCTION func1()  RETURNING INT;
   DEFINE GLOBAL gvar INT DEFAULT 2;
   LET gvar = gvar + 1;
   RETURN gvar;
END FUNCTION;

CREATE FUNCTION func2()  RETURNING INT;
   DEFINE GLOBAL gvar INT DEFAULT 5;
   LET gvar = gvar + 1;
   RETURN gvar;
END FUNCTION;
```

Although you must define a global variable with a default value, the variable is only set to the default the first time you use it. If you execute the two functions in the following figure in the order given, the value of **gvar** would be `4`.

Figure 420. Global variable default values.

```
EXECUTE FUNCTION func1();
EXECUTE FUNCTION func2();
```

But if you execute the functions in the opposite order, as the following figure shows, the value of **gvar** would be `7`.

Figure 421. Global variable default values.

```
EXECUTE FUNCTION func2();
EXECUTE FUNCTION func1();
```

For more information, see Executing routines on page 1804.

Assign values to variables

Within an SPL routine, use the LET statement to assign values to the variables you have already defined.

If you do not assign a value to a variable, either by an argument passed to the routine or by a LET statement, the variable has an undefined value.

An undefined value is different from a NULL value. If you attempt to use a variable with an undefined value within the SPL routine, you receive an error.

You can assign a value to a routine variable in any of the following ways:

- Use a LET statement.
- Use a SELECT INTO statement.
- Use a CALL statement with a procedure that has a RETURNING clause.
- Use an EXECUTE PROCEDURE INTO or EXECUTE FUNCTION INTO statement.

The LET statement

With a LET statement, you can use one or more variable names with an equal (=) sign and a valid expression or function name. Each example in the following figure is a valid LET statement.

Figure 422. Valid LET statements.

```
LET a = 5;
LET b = 6; LET c = 10;
LET a,b = 10,c+d;
LET a,b = (SELECT cola,colb
     FROM tab1 WHERE cola=10);
LET d = func1(x,y);
```

HCL Informix® allows you to assign a value to an opaque-type variable, a row-type variable, or a field of a row type. You can also return the value of an external function or another SPL function to an SPL variable.

Suppose you define the named row types **zip_t** and **address_t**, as Figure 409: Named and unnamed row variables. on page 1768 shows. Anytime you define a row-type variable, you must initialize the variable before you can use it. The following figure shows how you might define and initialize a row-type variable. You can use any row-type value to initialize the variable.

Figure 423. Define and initialize a row-type variable.

```
DEFINE a address_t;
LET a = ROW ('A Street', 'Nowhere', 'AA',
         ROW(NULL, NULL))::address_t
```

After you define and initialize the row-type variable, you can write the LET statements that the following figure shows.

Figure 424. Write the LET statements.

```
LET a.zip.z_code = 32601;
LET a.zip.z_suffix = 4555;
   -- Assign values to the fields of address_t
```

Tip: Use dot notation in the form **variable.field** or **variable.field.field** to access the fields of a row type, as Handle row-type data on page 1785 describes.

Suppose you define an opaque-type **point** that contains two values that define a two-dimensional point, and the text representation of the values is **'(x,y)'**. You might also have a function circum() that calculates the circumference of a circle, given the point **'(x,y)'** and a radius **r**.

If you define an opaque-type **center** that defines a point as the center of a circle, and a function circum() that calculates the circumference of a circle, based on a point and the radius, you can write variable declarations for each. In the following figure, **c** is an opaque type variable and **d** holds the value that the external function circum() returns.

Figure 425. Writing variable declarations.

```
DEFINE c point;
DEFINE r REAL;
DEFINE d REAL;

LET c = '(29.9,1.0)' ;
   -- Assign a value to an opaque type variable

LET d = circum( c, r );
   -- Assign a value returned from circum()
```

The *HCL® Informix® Guide to SQL: Syntax* describes in detail the syntax of the LET statement.

Other ways to assign values to variables

You can use the SELECT statement to fetch a value from the database and assign it directly to a variable, as the following figure shows.

Figure 426. Fetch a value from the database and assign it directly to a variable.

```
SELECT fname, lname INTO a, b FROM customer
   WHERE customer_num = 101
```

Use the CALL or EXECUTE PROCEDURE statements to assign values returned by an SPL function or an external function to one or more SPL variables. You might use either of the statements in the following figure to return the full name and address from the SPL function read_address into the specified SPL variables.

Figure 427. Return the full name and address from the SPL function.

```
EXECUTE FUNCTION read_address('Smith')
   INTO p_fname, p_lname, p_add, p_city, p_state,
        p_zip;

CALL read_address('Smith')
   RETURNING p_fname, p_lname, p_add, p_city,
             p_state, p_zip;
```

Expressions in SPL routines

You can use any SQL expression in an SPL routine, except for an aggregate expression. The *HCL® Informix® Guide to SQL: Syntax* provides the complete syntax and descriptions for SQL expressions.

The following examples contain SQL expressions:

```
var1
var1 + var2 + 5
read_address('Miller')
read_address(lastname = 'Miller')
get_duedate(acct_num) + 10 UNITS DAY
       fname[1,5] || ''|| lname '(415)' || get_phonenum(cust_name)
```

Writing the statement block

Every SPL routine has at least one statement block, which is a group of SQL and SPL statements between the CREATE statement and the END statement. You can use any SPL statement or any allowed SQL statement within a statement block. For a list of SQL statements that are not allowed within an SPL statement block, see the description of the statement block segment in the *HCL® Informix® Guide to SQL: Syntax*.

Implicit and explicit statement blocks

In an SPL routine, the *implicit statement block* extends from the end of the CREATE statement to the beginning of the END statement. You can also define an *explicit statement block*, which starts with a BEGIN statement and ends with an END statement, as the following figure shows.

Figure 428. Explicit statement block.

```
BEGIN
   DEFINE distance INT;
   LET distance = 2;
END
```

The explicit statement block allows you to define variables or processing that are valid only within the statement block. For example, you can define or redefine variables, or handle exceptions differently, for just the scope of the explicit statement block.

The SPL function in the following figure has an explicit statement block that redefines a variable defined in the implicit block.

Figure 429. An explicit statement block that redefines a variable defined in the implicit block.

```
CREATE FUNCTION block_demo()
   RETURNING INT;
      DEFINE distance INT;
   LET distance = 37;
   BEGIN
      DEFINE distance INT;
      LET distance = 2;
   END
   RETURN distance;

END FUNCTION;
```

In this example, the implicit statement block defines the variable **distance** and gives it a value of `37`. The explicit statement block defines a different variable named **distance** and gives it a value of `2`. However, the RETURN statement returns the value stored in the first **distance** variable, or `37`.

The FOREACH loop

A FOREACH loop defines a *cursor*, a specific identifier that points to one item in a group, whether a group of rows or the elements in a collection.

The FOREACH loop declares and opens a cursor, fetches rows from the database, works on each item in the group, and then closes the cursor. You must declare a cursor if a SELECT, EXECUTE PROCEDURE, or EXECUTE FUNCTION statement might return more than one row. After you declare the cursor, you place the SELECT, EXECUTE PROCEDURE, or EXECUTE FUNCTION statement within it.

An SPL routine that returns a group of rows is called a *cursor routine* because you must use a cursor to access the data it returns. An SPL routine that returns no value, a single value, or any other value that does not require a cursor is called a *noncursor routine*. The FOREACH loop declares and opens a cursor, fetches rows or a collection from the database, works on each item in the group, and then closes the cursor. You must declare a cursor if a SELECT, EXECUTE PROCEDURE, or EXECUTE FUNCTION statement might return more than one row or a collection. After you declare the cursor, you place the SELECT, EXECUTE PROCEDURE, or EXECUTE FUNCTION statement within it.

In a FOREACH loop, you can use an EXECUTE FUNCTION or SELECT INTO statement to execute an external function that is an iterator function.

The FOREACH loop to define cursors

A FOREACH loop begins with the FOREACH keyword and ends with END FOREACH. Between FOREACH and END FOREACH, you can declare a cursor or use EXECUTE PROCEDURE or EXECUTE FUNCTION. The two examples in the following figure show the structure of FOREACH loops.

Figure 430. Structure of FOREACH loops.

```
FOREACH cursor FOR
   SELECT column INTO variable FROM table
. . .
END FOREACH;

FOREACH
   EXECUTE FUNCTION name() INTO variable;
END FOREACH;
```

The following figure creates a routine that uses a FOREACH loop to operate on the **employee** table.

Figure 431. A FOREACH loop that operates on the employee table.

```
CREATE_PROCEDURE increase_by_pct( pct INTEGER )
   DEFINE s INTEGER;

   FOREACH sal_cursor FOR
      SELECT salary INTO s FROM employee
         WHERE salary > 35000
      LET s = s + s * ( pct/100 );
      UPDATE employee SET salary = s
         WHERE CURRENT OF sal_cursor;
   END FOREACH;

END PROCEDURE;
```

The routine in preceding figure performs these tasks within the FOREACH loop:

- Declares a cursor
- Selects one **salary** value at a time from **employee**
- Increases the salary by a percentage
- Updates **employee** with the new salary
- Fetches the next salary value

The SELECT statement is placed within a cursor because it returns all the salaries in the table greater than `35000`.

The WHERE CURRENT OF clause in the UPDATE statement updates only the row on which the cursor is currently positioned, and sets an *update cursor* on the current row. An update cursor places an update lock on the row so that no other user can update the row until your update occurs.

An SPL routine will set an update cursor automatically if an UPDATE or DELETE statement within the FOREACH loop uses the WHERE CURRENT OF clause. If you use WHERE CURRENT OF, you must explicitly reference the cursor within the FOREACH statement. If you are using an update cursor, you can add a BEGIN WORK statement before the FOREACH statement and a COMMIT WORK statement after END FOREACH, as the following figure shows.

Figure 432. Set an update cursor automatically.

```
BEGIN WORK;
   FOREACH sal_cursor FOR
      SELECT salary INTO s FROM employee WHERE salary > 35000;
      LET s = s + s * ( pct/100 );
      UPDATE employee SET salary = s WHERE CURRENT OF sal_cursor
   END FOREACH;
COMMIT WORK;
```

For each iteration of the FOREACH loop in the preceding figure, a new lock is acquired (if you use row level locking). The COMMIT WORK statement releases all of the locks (and commits all of the updated rows as a single transaction) after the last iteration of the FOREACH loop.

To commit an updated row after each iteration of the loop, you must open the cursor WITH HOLD, and include the BEGIN WORK and COMMIT WORK statements *within* the FOREACH loop, as in the following SPL routine.

Figure 433. Committing an updated row after each iteration of the loop.

```
CREATE PROCEDURE serial_update();
    DEFINE p_col2 INT;
    DEFINE i INT;
    LET i = 1;
    FOREACH cur_su WITH HOLD FOR
        SELECT col2 INTO p_col2 FROM customer WHERE 1=1
        BEGIN WORK;
            UPDATE customer SET customer_num = p_col2 WHERE CURRENT OF cur_su;
        COMMIT WORK;
        LET i = i + 1;
    END FOREACH;
END PROCEDURE;
```

SPL routine serial_update() commits each row as a separate transaction.

Restriction for FOREACH loops

Within a FOREACH loop, the SELECT query must complete execution before any DELETE, INSERT, or UPDATE operation that changes the data set of the SELECT cursor. One way to ensure that the SELECT query completes, use an ORDER BY clause in the SELECT statement. The ORDER BY clause creates an index on the columns and prevents errors caused by UPDATE, INSERT, DELETE statements modifying the query results of the SELECT statement in the same FOREACH loop

An IF - ELIF - ELSE structure

The following SPL routine uses an IF - ELIF - ELSE structure to compare the two arguments that the routine accepts.

Figure 434. An IF - ELIF - ELSE structure to compare two arguments.

```
CREATE FUNCTION str_compare( str1 CHAR(20), str2 CHAR(20))
   RETURNING INTEGER;

   DEFINE result INTEGER;

   IF str1 > str2 THEN
      LET result = 1;
   ELIF str2 > str1 THEN
      LET result = -1;
   ELSE
      LET result = 0;
   END IF
   RETURN result;
END FUNCTION;
```

Suppose you define a table named **manager** with the columns that the following figure shows.

Figure 435. Define the manager table.

```
CREATE TABLE manager
(
   mgr_name   VARCHAR(30),
   department VARCHAR(12),
   dept_no    SMALLINT,
   direct_reports SET( VARCHAR(30) NOT NULL ),
   projects LIST( ROW ( pro_name VARCHAR(15),
   pro_members SET( VARCHAR(20) NOT NULL ) )
                   NOT NULL),
   salary     INTEGER,
);
```

The following SPL routine uses an IF - ELIF - ELSE structure to check the number of elements in the SET in the **direct_reports** column and call various external routines based on the results.

Figure 436. An IF - ELIF - ELSE structure to check the number of elements in the SET.

```
CREATE FUNCTION checklist( d SMALLINT )
   RETURNING VARCHAR(30), VARCHAR(12), INTEGER;

   DEFINE name VARCHAR(30);
   DEFINE dept VARCHAR(12);
   DEFINE num INTEGER;

   SELECT mgr_name, department,
          CARDINALITY(direct_reports)
      FROM manager INTO name, dept, num
      WHERE dept_no = d;
   IF num > 20 THEN
      EXECUTE FUNCTION add_mgr(dept);
   ELIF num = 0 THEN
      EXECUTE FUNCTION del_mgr(dept);
   ELSE
      RETURN name, dept, num;
   END IF;

END FUNCTION;
```

The cardinality() function counts the number of elements that a collection contains. For more information, see Cardinality function on page 1603.

An IF - ELIF - ELSE structure in an SPL routine has up to the following four parts:

- An IF THEN condition

  If the condition following the IF statement is TRUE, the routine executes the statements in the IF block. If the condition is false, the routine evaluates the ELIF condition.

  The expression in an IF statement can be any valid condition, as the Condition segment of the *HCL® Informix® Guide to SQL: Syntax* describes. For the complete syntax and a detailed discussion of the IF statement, see the *HCL® Informix® Guide to SQL: Syntax*.

- One or more ELIF conditions (optional)

  The routine evaluates the ELIF condition only if the IF condition is false. If the ELIF condition is true, the routine executes the statements in the ELIF block. If the ELIF condition is false, the routine either evaluates the next ELIF block or executes the ELSE statement.

- An ELSE condition (optional)

  The routine executes the statements in the ELSE block if the IF condition and all of the ELIF conditions are false.

- An END IF statement

  The END IF statement ends the statement block.

Add WHILE and FOR loops

Both the WHILE and FOR statements create execution loops in SPL routines. A WHILE loop starts with a WHILE *condition*, executes a block of statements as long as the condition is true, and ends with END WHILE.

The following figure shows a valid WHILE condition. The routine executes the WHILE loop as long as the condition specified in the WHILE statement is true.

Figure 437. Routine to execute the WHILE loop as long as the condition specified in the WHILE statement is true.

```
CREATE PROCEDURE test_rows( num INT )

   DEFINE i INTEGER;
   LET i = 1;

   WHILE i < num
      INSERT INTO table1 (numbers) VALUES (i);
      LET i = i + 1;
   END WHILE;

END PROCEDURE;
```

The SPL routine in the previous figure accepts an integer as an argument and then inserts an integer value into the **numbers** column of **table1** each time it executes the WHILE loop. The values inserted start at `1` and increase to `num - 1`.

Be careful that you do not create an endless loop, as the following figure shows.

Figure 438. Routine to accept an integer as an argument and then insert an integer value into the numbers column.

```
CREATE PROCEDURE endless_loop()

   DEFINE i INTEGER;
   LET i = 1;
   WHILE ( 1 = 1 )            -- don't do this!
      LET i = i + 1;
      INSERT INTO table1 VALUES (i);
   END WHILE;

END PROCEDURE;
```

A FOR loop extends from a FOR statement to an END FOR statement and executes for a specified number of iterations, which are defined in the FOR statement. The following figure shows several ways to define the iterations in the FOR loop.

For each iteration of the FOR loop, the iteration variable (declared as *i* in the examples that follow) is reset, and the statements within the loop are executed with the new value of the variable.

Figure 439. Defining iterations in the FOR loop.

```
FOR i = 1 TO 10
. . .
END FOR;

FOR i = 1 TO 10 STEP 2
. . .
END FOR;

FOR i IN (2,4,8,14,22,32)
. . .
END FOR;

FOR i IN (1 TO 20 STEP 5, 20 to 1 STEP -5, 1,2,3,4,5)
. . .
END FOR:
```

In the first example, the SPL procedure executes the FOR loop as long as *i* is between `1` and `10`, inclusive. In the second example, *i* steps from `1` to `3`, `5`, `7`, and so on, but never exceeds `10`. The third example checks whether *i* is within a defined set of values. In the fourth example, the SPL procedure executes the loop when *i* is `1`, `6`, `11`, `16`, `20`, `15`, `10`, `5`, `1`, `2`, `3`, `4`, or `5`-in other words, 13 times.

Tip: The main difference between a WHILE loop and a FOR loop is that a FOR loop is guaranteed to finish, but a WHILE loop is not. The FOR statement specifies the exact number of times the loop executes, unless a statement causes the routine to exit the loop. With WHILE, it is possible to create an endless loop.

Exit a loop

In a FOR, FOREACH, LOOP, or WHILE loop that has no label, you can use the CONTINUE or EXIT statement to control the execution of the loop.

- CONTINUE causes the routine to skip the statements in the rest of the loop and move to the next iteration of the FOR, LOOP, or WHILE statement.
- EXIT ends the loop and causes the routine to continue executing with the first statement following the END FOR, the END LOOP, or the END WHILE keywords.

Remember that EXIT must be followed by the FOREACH keyword when it appears within a FOREACH statement that is the innermost loop of nested loop statements. EXIT can appear with no immediately following keyword when it appears within the FOR, LOOP, or WHILE statement, but an error is issued if you specify a keyword that does not match the loop statement from which the EXIT statement was issued. An error is also issued if EXIT appears outside the context of a loop statement.

For more information about loops in SPL routines, including labelled loops, see *HCL® Informix® Guide to SQL: Syntax*.

The following figure shows examples of CONTINUE and EXIT within a FOR loop.

Figure 440. Examples of CONTINUE and EXIT within a FOR loop.

```
FOR i = 1 TO 10
   IF i = 5 THEN
      CONTINUE FOR;
. . .
   ELIF i = 8 THEN
      EXIT FOR;
   END IF;

END FOR;
```

Tip: You can use CONTINUE and EXIT to improve the performance of SPL routines so that loops do not execute unnecessarily.

Return values from an SPL function

SPL functions can return one or more values. To have your SPL function return values, you need to include the following two parts:

1. Write a RETURNING clause in the CREATE PROCEDURE or CREATE FUNCTION statement that specifies the number of values to be returned and their data types.
2. In the body of the function, enter a RETURN statement that explicitly returns the values.

Tip: You can define a routine with the CREATE PROCEDURE statement that returns values, but in that case, the routine is actually a function. It is recommended that you use the CREATE FUNCTION statement when the routine returns values.

After you define a return clause (with a RETURNING statement), the SPL function can return values that match those specified in number and data type, or no values at all. If you specify a return clause, and the SPL routine returns no actual values, it is still considered a function. In that case, the routine returns a NULL value for each value defined in the return clause.

An SPL function can return variables, expressions, or the result of another function call. If the SPL function returns a variable, the function must first assign the variable a value by one of the following methods:

- A LET statement
- A default value
- A SELECT statement
- Another function that passes a value into the variable

Each value an SPL function returns can be up to 32 kilobytes long.

Important: The return value for an SPL function must be a specific data type. You cannot specify a generic row or generic collection data type as a return type.

Return a single value

The following figure shows how an SPL function can return a single value.

Figure 441. SPL function that returns a single value.

```
CREATE FUNCTION increase_by_pct(amt DECIMAL, pct DECIMAL)
   RETURNING DECIMAL;

   DEFINE result DECIMAL;

   LET result = amt + amt * (pct/100);

   RETURN result;

END FUNCTION;
```

The increase_by_pct function receives two arguments of DECIMAL value, an amount to be increased and a percentage by which to increase it. The return clause specifies that the function will return one DECIMAL value. The RETURN statement returns the DECIMAL value stored in **result**.

Return multiple values

An SPL function can return more than one value from a single row of a table. The following figure shows an SPL function that returns two column values from a single row of a table.

Figure 442. SPL function that returns two column values from a single row of a table.

```
CREATE FUNCTION birth_date( num INTEGER )
   RETURNING VARCHAR(30), DATE;

   DEFINE n VARCHAR(30);
   DEFINE b DATE;

   SELECT name, bdate INTO n, b FROM emp_tab
      WHERE emp_no = num;
   RETURN n, b;

END FUNCTION;
```

The function returns two values (a name and birthdate) to the calling routine from one row of the **emp_tab** table. In this case, the calling routine must be prepared to handle the VARCHAR and DATE values returned.

The following figure shows an SPL function that returns more than one value from more than one row.

Figure 443. SPL function that returns more than one value from more than one row.

```
CREATE FUNCTION birth_date_2( num INTEGER )
   RETURNING VARCHAR(30), DATE;
   DEFINE n VARCHAR(30);
   DEFINE b DATE;
   FOREACH cursor1 FOR
      SELECT name, bdate INTO n, b FROM emp_tab
         WHERE emp_no > num
      RETURN n, b WITH RESUME;
   END FOREACH;
END FUNCTION;
```

In preceding figure, the SELECT statement fetches two values from the set of rows whose employee number is higher than the number the user enters. The set of rows that satisfy the condition could contain one row, many rows, or zero rows. Because the SELECT statement can return many rows, it is placed within a cursor.

Tip: When a statement within an SPL routine returns no rows, the corresponding SPL variables are assigned NULL values.

The RETURN statement uses the WITH RESUME keywords. When RETURN WITH RESUME is executed, control is returned to the calling routine. But the next time the SPL function is called (by a FETCH or the next iteration of a cursor in the calling routine), all the variables in the SPL function keep their same values, and execution continues at the statement immediately following the RETURN WITH RESUME statement.

If your SPL routine returns multiple values, the calling routine must be able to handle the multiple values through a cursor or loop, as follows:

- If the calling routine is an SPL routine, it needs a FOREACH loop.
- If the calling routine is an ESQL/C program, it needs a cursor declared with the DECLARE statement.
- If the calling routine is an external routine, it needs a cursor or loop appropriate to the language in which the routine is written.

Important: The values returned by a UDR from external databases of a local server must be built-in data types or UDTs explicitly cast to built-in types or DISTINCT types based on built-in types and explicitly cast to built-in types. In addition, you must define the UDR and all the casts in the participating databases.

An example of SQL operations you can perform across databases follows:

```
database db1;
create table ltab1(lcol1 integer, lcol2 boolean, lcol3 lvarchar);
insert into ltab1 values(1, 't', "test string 1");

database db2;
create table rtab1(r1col1 boolean, r1col2 blob, r1col3 integer)
put r1col2 in (sbsp);
create table rtab2(r2col1 lvarchar, r2col2 clob) put r2col2 in (sbsp);
create table rtab3(r3col1 integer, r3col2 boolean,
r3col3 lvarchar, r3col4 circle);
```



```
create view rvw1 as select  * from rtab3;
```

(The example is a cross-database Insert.)

```
database db1;
create view lvw1 as select * from db2:rtab2;
insert into db2:rtab1 values('t',
filetoblob('blobfile', 'client', 'db2:rtab1', 'r1col2'), 100);
insert into db2:rtab2 values("inserted directly to rtab2",
filetoclob('clobfile', 'client', 'db2:rtab2', 'r2col2'));
insert into db2:rtab3 (r3col1, r3col2, r3col3)
select lcol1, lcol2, lcol3 from ltab1;
insert into db2:rvw1 values(200, 'f', "inserted via rvw1");
insert into lvw1 values ("inserted via lvw1", NULL);
```

Handle row-type data

In an SPL routine, you can use named ROW types and unnamed ROW types as parameter definitions, arguments, variable definitions, and return values. For information about how to declare a ROW variable in SPL, see Declare row-type variables on page 1767.

The following figure defines a row type **salary_t** and an **emp_info** table, which are the examples that this section uses.

Figure 444. Define a row type salary_t and an emp_info table

```
CREATE ROW TYPE salary_t(base MONEY(9,2), bonus MONEY(9,2))

CREATE TABLE emp_info (emp_name VARCHAR(30), salary salary_t);
```

The **emp_info** table has columns for the employee name and salary information.

Precedence of dot notation

With HCL Informix®, a value that uses dot notation (as in **proj.name**) in an SQL statement in an SPL routine is interpreted as having one of three meanings, in the following order of precedence:

1. *variable.field*
2. *column.field*
3. *table.column*

In other words, the expression **proj.name** is first evaluated as *variable.field.* If the routine does not find a variable **proj**, it evaluates the expression as *column.field*. If the routine does not find a column **proj**, it evaluates the expression as *table.column*. (If the names cannot be resolved as identifiers of objects in the database or of variables or fields that were declared in the SPL routine, then an error is returned.)

Update a row-type expression

From within an SPL routine, you can use a ROW variable to update a row-type expression. The following figure shows an SPL procedure **emp_raise** that is used to update the **emp_info** table when an employee's base salary increases by a certain percentage.

Figure 445. SPL procedure used to update the emp_info table.

```
CREATE PROCEDURE emp_raise( name VARCHAR(30),
                  pct DECIMAL(3,2) )

   DEFINE row_var salary_t;

   SELECT salary INTO row_var FROM emp_info
      WHERE emp_name = name;

   LET row_var.base = row_var.base * pct;

   UPDATE emp_info SET salary = row_var
      WHERE emp_name = name;
END PROCEDURE;
```

The SELECT statement selects a row from the **salary** column of **emp_info table** into the ROW variable **row_var**.

The **emp_raise** procedure uses SPL dot notation to directly access the **base** field of the variable **row_var**. In this case, the dot notation means *variable.field*. The **emp_raise** procedure recalculates the value of **row_var.base** as `(row_var.base * pct)`. The procedure then updates the **salary** column of the **emp_info** table with the new **row_var** value.

Important: A row-type variable must be initialized as a row before its fields can be set or referenced. You can initialize a row-type variable with a SELECT INTO statement or LET statement.

Handle collections

A *collection* is a group of elements of the same data type, such as a SET, MULTISET, or LIST.

A table might contain a collection stored as the contents of a column or as a field of a ROW type within a column. A collection can be either simple or nested. A *simple collection* is a SET, MULTISET, or LIST of built-in, opaque, or distinct data types. A *nested collection* is a collection that contains other collections.

Collection data types

The following sections of the chapter rely on several different examples to show how you can manipulate collections in SPL programs.

The basics of handling collections in SPL programs are illustrated with the **numbers** table, as the following figure shows.

Figure 446. Handle collections in SPL programs.

```
CREATE TABLE numbers
(
   id INTEGER PRIMARY KEY,
   primes       SET( INTEGER NOT NULL ),
   evens        LIST( INTEGER NOT NULL ),
   twin_primes  LIST( SET( INTEGER NOT NULL )
                   NOT NULL )
```

The **primes** and **evens** columns hold simple collections. The **twin_primes** column holds a nested collection, a LIST of SETs. (Twin prime numbers are pairs of consecutive prime numbers whose difference is 2, such as 5 and 7, or 11 and 13. The **twin_primes** column is designed to allow you to enter such pairs.

Some examples in this chapter use the **polygons** table in the following figure to illustrate how to manipulate collections. The **polygons** table contains a collection to represent two-dimensional graphical data. For example, suppose that you define an opaque data type named **point** that has two double-precision values that represent the **x** and **y** coordinates of a two-dimensional point whose coordinates might be represented as `'1.0, 3.0'`. Using the **point** data type, you can create a table that contains a set of points that define a polygon.

Figure 447. Manipulate collections.

```
CREATE OPAQUE TYPE point ( INTERNALLENGTH = 8);

CREATE TABLE polygons
(
   id          INTEGER PRIMARY KEY,
   definition  SET( point NOT NULL )
);
```

The **definition** column in the **polygons** table contains a simple collection, a SET of **point** values.

Prepare for collection data types

Before you can access and handle an individual element of a simple or nested collection, you must perform the following tasks:

- Declare a collection variable to hold the collection.
- Declare an element variable to hold an individual element of the collection.
- Select the collection from the database into the collection variable.

After you take these initial steps, you can insert elements into the collection or select and handle elements that are already in the collection.

Each of these steps is explained in the following sections, using the **numbers** table as an example.

Tip: You can handle collections in any SPL routine.

Declare a collection variable

Before you can retrieve a collection from the database into an SPL routine, you must declare a collection variable. The following figure shows how to declare a collection variable to retrieve the **primes** column from the **numbers** table.

Figure 448. Declare a collection variable.

```
DEFINE p_coll SET( INTEGER NOT NULL );
```

The DEFINE statement declares a collection variable **p_coll**, whose type matches the data type of the collection stored in the **primes** column.

Declare an element variable

After you declare a collection variable, you declare an element variable to hold individual elements of the collection. The data type of the element variable must match the data type of the collection elements.

For example, to hold an element of the SET in the **primes** column, use an element variable declaration such as the one that the following figure shows.

Figure 449. An element variable declaration.

```
DEFINE p INTEGER;
```

To declare a variable that holds an element of the **twin_primes** column, which holds a nested collection, use a variable declaration such as the one that the following figure shows.

Figure 450. A variable declaration.

```
DEFINE s SET( INTEGER NOT NULL );
```

The variable **s** holds a SET of integers. Each SET is an element of the LIST stored in **twin_primes**.

Select a collection into a collection variable

After you declare a collection variable, you can fetch a collection into it. To fetch a collection into a collection variable, enter a SELECT INTO statement that selects the collection column from the database into the collection variable you have named.

For example, to select the collection stored in one row of the **primes** column of **numbers**, add a SELECT statement, such as the one that the following figure shows, to your SPL routine.

Figure 451. Add a SELECT statement to select the collection stored in one row.

```
SELECT primes INTO p_coll FROM numbers
   WHERE id = 220;
```

The WHERE clause in the SELECT statement specifies that you want to select the collection stored in just one row of **numbers**. The statement places the collection into the collection variable **p_coll**, which Figure 448: Declare a collection variable. on page 1788 declares.

The variable **p_coll** now holds a collection from the **primes** column, which could contain the value `SET {5,7,31,19,13}`.

Insert elements into a collection variable

After you retrieve a collection into a collection variable, you can insert a value into the collection variable. The syntax of the INSERT statement varies slightly, depending on the type of the collection to which you want to add values.

Insert into a SET or MULTISET

To insert into a SET or MULTISET stored in a collection variable, use an INSERT statement and follow the TABLE keyword with the collection variable, as the following figure shows.

Figure 452. Insert into a SET or MULTISET stored in a collection variable.

```
INSERT INTO TABLE(p_coll) VALUES(3);
```

The TABLE keyword makes the collection variable a collection-derived table. Collection-derived tables are described in the section Handle collections in SELECT statements on page 1645. The collection that the previous figure derives is a virtual table of one column, with each element of the collection representing a row of the table. Before the insert, consider **p_coll** as a virtual table that contains the rows (elements) that the following figure shows.

Figure 453. Virtual table elements.

```
5
7
31
19
13
```

After the insert, **p_coll** might look like the virtual table that the following figure shows.

Figure 454. Virtual table elements.

```
5
7
31
19
13
3
```

Because the collection is a SET, the new value is added to the collection, but the position of the new element is undefined. The same principle is true for a MULTISET.

Tip: You can only insert one value at a time into a simple collection.

Insert into a LIST

If the collection is a LIST, you can add the new element at a specific point in the LIST or at the end of the LIST. As with a SET or MULTISET, you must first define a collection variable and select a collection from the database into the collection variable.

The following figure shows the statements you need to define a collection variable and select a LIST from the **numbers** table into the collection variable.

Figure 455. Defining a collection variable and selecting a LIST.

```
DEFINE e_coll LIST(INTEGER NOT NULL);

SELECT evens INTO e_coll FROM numbers
   WHERE id = 99;
```

At this point, the value of **e_coll** might be `LIST {2,4,6,8,10}`. Because **e_coll** holds a LIST, each element has a numbered position in the list. To add an element at a specific point in a LIST, add an AT *position* clause to the INSERT statement, as the following figure shows.

Figure 456. Add an element at a specific point in a LIST.

```
INSERT AT 3 INTO TABLE(e_coll) VALUES(12);
```

Now the LIST in **e_coll** has the elements `{2,4,12,6,8,10}`, in that order.

The value you enter for the *position* in the AT clause can be a number or a variable, but it must have an INTEGER or SMALLINT data type. You cannot use a letter, floating-point number, decimal value, or expression.

Check the cardinality of a LIST collection

At times you might want to add an element at the end of a LIST. In this case, you can use the cardinality() function to find the number of elements in a LIST and then enter a position that is greater than the value cardinality() returns.

HCL Informix® allows you to use the cardinality() function with a collection that is stored in a column but not with a collection that is stored in a collection variable. In an SPL routine, you can check the cardinality of a collection in a column with a SELECT statement and return the value to a variable.

Suppose that in the **numbers** table, the **evens** column of the row whose **id** column is 99 still contains the collection `LIST {2,4,6,8,10}`. This time, you want to add the element `12` at the end of the LIST. You can do so with the SPL procedure **end_of_list**, as the following figure shows.

Figure 457. The end_of_list SPL procedure.

```
CREATE PROCEDURE end_of_list()

   DEFINE n SMALLINT;
   DEFINE list_var LIST(INTEGER NOT NULL);

   SELECT CARDINALITY(evens) FROM numbers INTO n
      WHERE id = 100;

   LET n = n + 1;

   SELECT evens INTO list_var FROM numbers
      WHERE id = 100;

   INSERT AT n INTO TABLE(list_var) VALUES(12);

END PROCEDURE;
```

In **end_of_list**, the variable **n** holds the value that cardinality() returns, that is, the count of the items in the LIST. The LET statement increments **n**, so that the INSERT statement can insert a value at the last position of the LIST. The SELECT statement selects the collection from one row of the table into the collection variable **list_var**. The INSERT statement inserts the element `12` at the end of the list.

Syntax of the VALUES clause

The syntax of the VALUES clause is different when you insert into an SPL collection variable from when you insert into a collection column. The syntax rules for inserting literals into collection variables are as follows:

- Use parentheses after the VALUES keyword to enclose the complete list of values.
- If you are inserting into a simple collection, you do not need to use a type constructor or brackets.
- If you are inserting into a nested collection, you need to specify a literal collection.

Select elements from a collection

Suppose you want your SPL routine to select elements from the collection stored in the collection variable, one at time, so that you can handle the elements.

To move through the elements of a collection, you first need to declare a cursor using a FOREACH statement, just as you would declare a cursor to move through a set of rows. The following figure shows the FOREACH and END FOREACH statements, with no statements between them yet.

Figure 458. FOREACH and END FOREACH statements.

```
FOREACH cursor1 FOR
. . .
END FOREACH
```

The FOREACH statement is described in The FOREACH loop on page 1776 and the *HCL® Informix® Guide to SQL: Syntax*.

The next topic, The collection query on page 1792, describes the statements that are omitted between the FOREACH and END FOREACH statements.

The examples in the following sections are based on the **polygons** table of Figure 447: Manipulate collections. on page 1787.

The collection query

After you declare the cursor between the FOREACH and END FOREACH statements, you enter a special, restricted form of the SELECT statement known as a *collection query*.

A collection query is a SELECT statement that uses the FROM TABLE keywords followed by the name of a collection variable. The following figure shows this structure, which is known as a *collection-derived table*.

Figure 459. Collection-derived table.

```
FOREACH cursor1 FOR

   SELECT * INTO pnt FROM TABLE(vertexes)
   . . .
END FOREACH
```

The SELECT statement uses the collection variable **vertexes** as a collection-derived table. You can think of a collection-derived table as a table of one column, with each element of the collection being a row of the table. For example, you can visualize the SET of four points stored in **vertexes** as a table with four rows, such as the one that the following figure shows.

Figure 460. Table with four rows.

```
'(3.0,1.0)'
'(8.0,1.0)'
'(3.0,4.0)'
'(8.0,4.0)'
```

After the first iteration of the FOREACH statement in the previous figure, the collection query selects the first element in **vertexes** and stores it in **pnt**, so that **pnt** contains the value '`(3.0,1.0)`'.

Tip: Because the collection variable **vertexes** contains a SET, not a LIST, the elements in **vertexes** have no defined order. In a real database, the value `'(3.0,1.0)'` might not be the first element in the SET.

Add the collection query to the SPL routine

Now you can add the cursor defined with FOREACH and the collection query to the SPL routine, as the following example shows.

Figure 461. Cursor defined with FOREACH and the collection query.

```
CREATE PROCEDURE shapes()

   DEFINE vertexes SET( point NOT NULL );
   DEFINE pnt point;

   SELECT definition INTO vertexes FROM polygons
      WHERE id = 207;

   FOREACH cursor1 FOR
      SELECT * INTO pnt FROM TABLE(vertexes)
   . . .
   END FOREACH
. . .
END PROCEDURE;
```

The statements shown above form the framework of an SPL routine that handles the elements of a collection variable. To decompose a collection into its elements, use a collection-derived table. After the collection is decomposed into its elements, the routine can access elements individually as rows of the collection-derived table. Now that you have selected one element in **pnt**, you can update or delete that element, as Update a collection element on page 1796 and Delete a collection element on page 1793 describe.

For the complete syntax of the collection query, see the SELECT statement in the *HCL® Informix® Guide to SQL: Syntax*. For the syntax of a collection-derived table, see the Collection-Derived Table segment in the *HCL® Informix® Guide to SQL: Syntax*.

Tip: If you are selecting from a collection that contains no elements or zero elements, you can use a collection query without declaring a cursor. However, if the collection contains more than one element and you do not use a cursor, you will receive an error message.

Attention: In the program fragment above, the database server would have issued a syntax error if the query (

```
SELECT * INTO pnt FROM TABLE(vertexes)
```

) within the FOREACH cursor definition had ended with a semicolon (`;`) character as a statement terminator. Here the END FOREACH keywords are the logical statement terminator.

Delete a collection element

After you select an individual element from a collection variable into an element variable, you can delete the element from the collection. For example, after you select a point from the collection variable **vertexes** with a collection query, you can remove the point from the collection.

The steps involved in deleting a collection element include:

1. Declare a collection variable and an element variable.
2. Select the collection from the database into the collection variable.

3. Declare a cursor so that you can select elements one at a time from the collection variable.
4. Write a loop or branch that locates the element that you want to delete.
5. Delete the element from the collection using a DELETE WHERE CURRENT OF statement that uses the collection variable as a collection-derived table.

The following figure shows a routine that deletes one of the four points in **vertexes**, so that the polygon becomes a triangle instead of a rectangle.

Figure 462. Routine that deletes one of the four points.

```
CREATE PROCEDURE shapes()

   DEFINE vertexes SET( point NOT NULL );
   DEFINE pnt point;

   SELECT definition INTO vertexes FROM polygons
      WHERE id = 207;

   FOREACH cursor1 FOR
      SELECT * INTO pnt FROM TABLE(vertexes)
      IF pnt = '(3,4)' THEN
            -- calls the equals function that
            -- compares two values of point type
         DELETE FROM TABLE(vertexes)
            WHERE CURRENT OF cursor1;
         EXIT FOREACH;
      ELSE
         CONTINUE FOREACH;
      END IF;
   END FOREACH
. . .
END PROCEDURE;
```

In previous figure, the FOREACH statement declares a cursor. The SELECT statement is a collection-derived query that selects one element at a time from the collection variable **vertexes** into the element variable **pnt**.

The IF THEN ELSE structure tests the value currently in **pnt** to see if it is the point `'(3,4)'`. Note that the expression `pnt = '(3,4)'` calls the instance of the equal() function defined on the point data type. If the current value in **pnt** is `'(3,4)'`, the DELETE statement deletes it, and the EXIT FOREACH statement exits the cursor.

Tip: Deleting an element from a collection stored in a collection variable does not delete it from the collection stored in the database. After you delete the element from a collection variable, you must update the collection stored in the database with the new collection. For an example that shows how to update a collection column, see Update the collection in the database on page 1795.

The syntax for the DELETE statement is described in the *HCL® Informix® Guide to SQL: Syntax*.

Update the collection in the database

After you change the contents of a collection variable in an SPL routine (by deleting, updating, or inserting an element), you must update the database with the new collection.

To update a collection in the database, add an UPDATE statement that sets the collection column in the table to the contents of the updated collection variable. For example, the UPDATE statement in the following figure shows how to update the **polygons** table to set the **definition** column to the new collection stored in the collection variable **vertexes**.

Figure 463. Update a collection in the database.

```
CREATE PROCEDURE shapes()

   DEFINE vertexes SET(point NOT NULL);
   DEFINE pnt point;

   SELECT definition INTO vertexes FROM polygons
      WHERE id = 207;

   FOREACH cursor1 FOR
      SELECT * INTO pnt FROM TABLE(vertexes)
      IF pnt = '(3,4)' THEN
            -- calls the equals function that
            -- compares two values of point type
         DELETE FROM TABLE(vertexes)
            WHERE CURRENT OF cursor1;
         EXIT FOREACH;
      ELSE
         CONTINUE FOREACH;
      END IF;
   END FOREACH

   UPDATE polygons SET definition = vertexes
      WHERE id = 207;

END PROCEDURE;
```

Now the **shapes()** routine is complete. After you run **shapes()**, the collection stored in the row whose ID column is `207` is updated so that it contains three values instead of four.

You can use the **shapes()** routine as a framework for writing other SPL routines that manipulate collections.

The elements of the collection now stored in the **definition** column of row `207` of the **polygons** table are listed as follows:

```
'(3,1)'
'(8,1)'
'(8,4)'
```

Delete the entire collection

If you want to delete all the elements of a collection, you can use a single SQL statement. You do not need to declare a cursor. To delete an entire collection, you must perform the following tasks:

- Define a collection variable.
- Select the collection from the database into a collection variable.
- Enter a DELETE statement that uses the collection variable as a collection-derived table.
- Update the collection from the database.

The following figure shows the statements that you might use in an SPL routine to delete an entire collection.

Figure 464. SPL routine to delete an entire collection.

```
DEFINE vertexes SET( INTEGER NOT NULL );

SELECT definition INTO vertexes FROM polygons
   WHERE id = 207;

DELETE FROM TABLE(vertexes);

UPDATE polygons SET definition = vertexes
   WHERE id = 207;
```

This form of the DELETE statement deletes the entire collection in the collection variable **vertexes**. You cannot use a WHERE clause in a DELETE statement that uses a collection-derived table.

After the UPDATE statement, the **polygons** table contains an empty collection where the **id** column is equal to `207`.

The syntax for the DELETE statement is described in the *HCL® Informix® Guide to SQL: Syntax*.

Update a collection element

You can update a collection element by accessing the collection within a cursor just as you select or delete an individual element.

If you want to update the collection `SET{100, 200, 300, 500}` to change the value `500` to `400`, retrieve the SET from the database into a collection variable and then declare a cursor to move through the elements in the SET, as the following figure shows.

Figure 465. Update the collection element.

```
DEFINE s SET(INTEGER NOT NULL);
DEFINE n INTEGER;

SELECT numbers INTO s FROM orders
   WHERE order_num = 10;

FOREACH cursor1 FOR
   SELECT * INTO n FROM TABLE(s)
   IF ( n == 500 ) THEN
      UPDATE TABLE(s)(x)
         SET x = 400 WHERE CURRENT OF cursor1;
      EXIT FOREACH;
   ELSE
      CONTINUE FOREACH;
   END IF;
END FOREACH
```

The UPDATE statement uses the collection variable **s** as a collection-derived table. To specify a collection-derived table, use the TABLE keyword. The value `(x)` that follows `(s)` in the UPDATE statement is a *derived column*, a column name you supply because the SET clause requires it, even though the collection-derived table does not have columns.

Think of the collection-derived table as having one row and looking something like the following example:

```
100      200      300      500
```

In this example, `x` is a fictitious column name for the column that contains the value `500`. You only specify a derived column if you are updating a collection of built-in, opaque, distinct, or collection type elements. If you are updating a collection of row types, use a field name instead of a derived column, as Update a collection of row types on page 1798 describes.

Update a collection with a variable

You can also update a collection with the value stored in a variable instead of a literal value.

The SPL procedure in the following figure uses statements that are similar to the ones that Figure 465: Update the collection element. on page 1796 shows, except that this procedure updates the SET in the **direct_reports** column of the **manager** table with a variable, rather than with a literal value. Figure 435: Define the manager table. on page 1779 defines the **manager** table.

Figure 466. Update a collection with a variable.

```
CREATE PROCEDURE new_report(mgr VARCHAR(30),
   old VARCHAR(30), new VARCHAR(30) )

   DEFINE s SET (VARCHAR(30) NOT NULL);
   DEFINE n VARCHAR(30);

   SELECT direct_reports INTO s FROM manager
      WHERE mgr_name = mgr;

   FOREACH cursor1 FOR
      SELECT * INTO n FROM TABLE(s)
      IF ( n == old ) THEN
         UPDATE TABLE(s)(x)
            SET x = new WHERE CURRENT OF cursor1;
         EXIT FOREACH;
      ELSE
         CONTINUE FOREACH;
      END IF;
   END FOREACH

   UPDATE manager SET mgr_name = s
      WHERE mgr_name = mgr;

END PROCEDURE;
```

The UPDATE statement nested in the FOREACH loop uses the collection- derived table **s** and the derived column **x**. If the current value of *n* is the same as **old**, the UPDATE statement changes it to the value of **new**. The second UPDATE statement stores the new collection in the **manager** table.

Update the entire collection

If you want to update all the elements of a collection to the same value, or if the collection contains only one element, you do not need to use a cursor. The statements in the following figure show how you can retrieve the collection into a collection variable and then update it with one statement.

Figure 467. Retrieve and update the collection.

```
DEFINE s SET (INTEGER NOT NULL);

SELECT numbers INTO s FROM orders
   WHERE order_num = 10;

UPDATE TABLE(s)(x) SET x = 0;

UPDATE orders SET numbers = s
   WHERE order_num = 10;
```

The first UPDATE statement in this example uses a derived column named **x** with the collection-derived table **s** and gives all the elements in the collection the value `0`. The second UPDATE statement stores the new collection in the database.

Update a collection of row types

To update a collection of ROW types, you can take these steps:

1. Declare a collection variable whose field data types match those of the ROW types in the collection.
2. Set the individual fields of the collection variable to the correct data values for the ROW type.
3. For each ROW type, update the entire row of the collection derived table using the collection variable.

The **manager** table in Figure 435: Define the manager table. on page 1779 has a column named **projects** that contains a LIST of ROW types with the definition that the following figure shows.

Figure 468. LIST of ROW types definition.

```
projects   LIST( ROW( pro_name VARCHAR(15),
           pro_members SET(VARCHAR(20) NOT NULL) ) NOT NULL)
```

To access the ROW types in the LIST, declare a cursor and select the LIST into a collection variable. After you retrieve each ROW type value in the **projects** column, however, you cannot update the **pro_name** or **pro_members** fields individually. Instead, for each ROW value that needs to be updated in the collection, you must replace the entire ROW with values from a collection variable that include the new field values, as the following figure shows.

Figure 469. Access the ROW types in the LIST.

```
CREATE PROCEDURE update_pro( mgr VARCHAR(30),
   pro VARCHAR(15) )

   DEFINE p LIST(ROW(a VARCHAR(15), b SET(VARCHAR(20)
          NOT NULL) ) NOT NULL);
   DEFINE r ROW(p_name VARCHAR(15), p_member SET(VARCHAR(20) NOT NULL) );
   LET r = ROW("project", "SET{'member'}");

SELECT projects INTO p FROM manager
      WHERE mgr_name = mgr;

   FOREACH cursor1 FOR
      SELECT * INTO r FROM TABLE(p)
      IF (r.p_name == 'Zephyr') THEN
         LET r.p_name = pro;
         UPDATE TABLE(p)(x) SET x = r
            WHERE CURRENT OF cursor1;
         EXIT FOREACH;
      END IF;
   END FOREACH

   UPDATE manager SET projects = p
      WHERE mgr_name = mgr;

END PROCEDURE;
```

Before you can use a row-type variable in an SPL program, you must initialize the row variable with a LET statement or a SELECT INTO statement. The UPDATE statement nested in the FOREACH loop of the previous figure sets the **pro_name** field of the row type to the value supplied in the variable **pro**.

Tip: To update a value in a SET in the **pro_members** field of the ROW type, declare a cursor and use an UPDATE statement with a derived column, as Update a collection element on page 1796 explains.

Update a nested collection

If you want to update a collection of collections, you must declare a cursor to access the outer collection and then declare a nested cursor to access the inner collection.

For example, suppose that the **manager** table has an additional column, **scores**, which contains a LIST whose element type is a MULTISET of integers, as the following figure shows.

Figure 470. Update a collection of collections.

```
scores         LIST(MULTISET(INT NOT NULL) NOT NULL);
```

To update a value in the MULTISET, declare a cursor that moves through each value in the LIST and a nested cursor that moves through each value in the MULTISET, as the following figure shows.

Figure 471. Update a value in the MULTISET.

```
CREATE FUNCTION check_scores ( mgr VARCHAR(30) )
   SPECIFIC NAME nested;
   RETURNING INT;

   DEFINE l LIST( MULTISET( INT NOT NULL ) NOT NULL );
   DEFINE m MULTISET( INT NOT NULL );
   DEFINE n INT;
   DEFINE c INT;

   SELECT scores INTO l FROM manager
      WHERE mgr_name = mgr;

   FOREACH list_cursor FOR
      SELECT * FROM TABLE(l) INTO m;

      FOREACH set_cursor FOR
         SELECT * FROM TABLE(m) INTO n;
         IF (n == 0) THEN
            DELETE FROM TABLE(m)
               WHERE CURRENT OF set_cursor;
         ENDIF;
      END FOREACH;
      LET c = CARDINALITY(m);
      RETURN c WITH RESUME;
   END FOREACH

END FUNCTION
   WITH LISTING IN '/tmp/nested.out';
```

The SPL function selects each MULTISET in the **scores** column into **l**, and then each value in the MULTISET into **m**. If a value in **m** is `0`, the function deletes it from the MULTISET. After the values of `0` are deleted, the function counts the remaining elements in each MULTISET and returns an integer.

Tip: Because this function returns a value for each MULTISET in the LIST, you must use a cursor to enclose the EXECUTE FUNCTION statement when you execute the function.

Insert into a collection

You can insert a value into a collection without declaring a cursor. If the collection is a SET or MULTISET, the value is added to the collection but the position of the new element is undefined because the collection has no particular order. If the value is a LIST, you can add the new element at a specific point in the LIST or at the end of the LIST.

In the **manager** table, the **direct_reports** column contains collections of SET type, and the **projects** column contains a LIST. To add a name to the SET in the **direct_reports** column, use an INSERT statement with a collection-derived table, as the following figure shows.

Figure 472. Insert a value into a collection.

```
CREATE PROCEDURE new_emp( emp VARCHAR(30), mgr VARCHAR(30) )

   DEFINE r SET(VARCHAR(30) NOT NULL);

   SELECT direct_reports INTO r FROM manager
      WHERE mgr_name = mgr;

   INSERT INTO TABLE (r) VALUES(emp);

   UPDATE manager SET direct_reports = r
      WHERE mgr_name = mgr;

END PROCEDURE;
```

This SPL procedure takes an employee name and a manager name as arguments. The procedure then selects the collection in the **direct_reports** column for the manager the user has entered, adds the employee name the user has entered, and updates the **manager** table with the new collection.

The INSERT statement in the previous figure inserts the new employee name that the user supplies into the SET contained in the collection variable **r**. The UPDATE statement then stores the new collection in the **manager** table.

Notice the syntax of the VALUES clause. The syntax rules for inserting literal data and variables into collection variables are as follows:

- Use parentheses after the VALUES keyword to enclose the complete list of values.
- If the collection is SET, MULTISET, or LIST, use the type constructor followed by brackets to enclose the list of values to be inserted. In addition, the collection value must be enclosed in quotes.

  ```
  VALUES( "SET{ 1,4,8,9 }" )
  ```

- If the collection contains a row type, use ROW followed by parentheses to enclose the list of values to be inserted:

  ```
  VALUES( ROW( 'Waters', 'voyager_project' ) )
  ```

- If the collection is a nested collection, nest keywords, parentheses, and brackets according to how the data type is defined:

  ```
  VALUES( "SET{ ROW('Waters', 'voyager_project'),
              ROW('Adams', 'horizon_project') }")
  ```

For more information on inserting values into collections, see Modify data on page 1658.

Insert into a nested collection

If you want to insert into a nested collection, the syntax of the VALUES clause changes. Suppose, for example, that you want to insert a value into the **twin_primes** column of the **numbers** table that Figure 446: Handle collections in SPL programs. on page 1787 shows.

With the **twin_primes** column, you might want to insert a SET into the LIST or an element into the inner SET. The following sections describe each of these tasks.

Insert a collection into the outer collection

Inserting a SET into the LIST is similar to inserting a single value into a simple collection.

To insert a SET into the LIST, declare a collection variable to hold the LIST and select the entire collection into it. When you use the collection variable as a collection-derived table, each SET in the LIST becomes a row in the table. You can then insert another SET at the end of the LIST or at a specified point.

For example, the **twin_primes** column of one row of numbers might contain the following LIST, as the following figure shows.

Figure 473. Sample LIST.

```
LIST( SET{3,5}, SET{5,7}, SET{11,13} )
```

If you think of the LIST as a collection-derived table, it might look similar to the following.

Figure 474. Thinking of the LIST as a collection-derived table.

```
{3,5}
{5,7}
{11,13}
```

You might want to insert the value `"SET{17,19}"` as a second item in the LIST. The statements in the following figure show how to do this.

Figure 475. Insert a value in the LIST.

```
CREATE PROCEDURE add_set()

   DEFINE l_var LIST( SET( INTEGER NOT NULL ) NOT NULL );

   SELECT twin_primes INTO l_var FROM numbers
      WHERE id = 100;

   INSERT AT 2 INTO TABLE (l_var) VALUES( "SET{17,19}" );

   UPDATE numbers SET twin_primes = l
      WHERE id = 100;

END PROCEDURE;
```

In the INSERT statement, the VALUES clause inserts the value `SET {17,19}` at the second position of the LIST. Now the LIST looks like the following figure.

Figure 476. LIST items.

```
{3,5}
{17,19}
{5,7}
{11,13}
```

You can perform the same insert by passing a SET to an SPL routine as an argument, as the following figure shows.

Figure 477. Passing a SET to an SPL routine as an argument.

```
CREATE PROCEDURE add_set( set_var SET(INTEGER NOT NULL),
   row_id INTEGER );

   DEFINE list_var LIST( SET(INTEGER NOT NULL) NOT NULL );
   DEFINE n SMALLINT;

   SELECT CARDINALITY(twin_primes) INTO n FROM numbers
      WHERE id = row_id;

   LET n = n + 1;

   SELECT twin_primes INTO list_var FROM numbers
      WHERE id = row_id;

   INSERT AT n INTO TABLE( list_var ) VALUES( set_var );

   UPDATE numbers SET twin_primes = list_var
      WHERE id = row_id;

END PROCEDURE;
```

In add_set(), the user supplies a SET to add to the LIST and an INTEGER value that is the **id** of the row in which the SET will be inserted.

Insert a value into the inner collection

In an SPL routine, you can also insert a value into the inner collection of a nested collection. In general, to access the inner collection of a nested collection and add a value to it, perform the following steps:

1. Declare a collection variable to hold the entire collection stored in one row of a table.
2. Declare an element variable to hold one element of the outer collection. The element variable is itself a collection variable.
3. Select the entire collection from one row of a table into the collection variable.
4. Declare a cursor so that you can move through the elements of the outer collection.
5. Select one element at a time into the element variable.
6. Use a branch or loop to locate the inner collection you want to update.
7. Insert the new value into the inner collection.
8. Close the cursor.
9. Update the database table with the new collection.

As an example, you can use this process on the **twin_primes** column of **numbers**. For example, suppose that **twin_primes** contains the values that the following figure shows, and you want to insert the value `18` into the last SET in the LIST.

Figure 478. The twin_primes list.

```
LIST( SET( {3,5}, {5,7}, {11,13}, {17,19} ) )
```

The following figure shows the beginning of a procedure that inserts the value.

Figure 479. Procedure that inserts the value.

```
CREATE PROCEDURE add_int()

   DEFINE list_var LIST( SET( INTEGER NOT NULL ) NOT NULL );
   DEFINE set_var SET( INTEGER NOT NULL );

   SELECT twin_primes INTO list_var FROM numbers
         WHERE id = 100;
```

So far, the **attaint** procedure has performed steps 1 on page 1803, 2 on page 1803, and 3 on page 1803. The first DEFINE statement declares a collection variable that holds the entire collection stored in one row of numbers.

The second DEFINE statement declares an element variable that holds an element of the collection. In this case, the element variable is itself a collection variable because it holds a SET. The SELECT statement selects the entire collection from one row into the collection variable, **list_var**.

The following figure shows how to declare a cursor so that you can move through the elements of the outer collection.

Figure 480. Declare a cursor to move through the elements of the outer collection.

```
FOREACH list_cursor FOR
   SELECT * INTO set_var FROM TABLE( list_var);

   FOREACH element_cursor FOR
```

Executing routines

You can execute an SPL routine or external routine in any of these ways:

- Using a stand-alone EXECUTE PROCEDURE or EXECUTE FUNCTION statement that you execute from
- Calling the routine explicitly from another SPL routine or an external routine
- Using the routine name with an expression in an SQL statement

An additional mechanism for executing routines supports only the **sysdbopen** and **sysdbclose** procedures, which the DBA can define. If a **sysdbopen** procedure whose owner matches the login identifier of a user exists in the database when the user connects to the database by the CONNECT or DATABASE statement, that routine is executed automatically. If no **sysdbopen** routine has an owner that matches the login identifier of the user, but a PUBLIC**.sysdbopen** routine exists, that routine is executed. This automatic invocation enables the DBA to customize the session environment for users at connection time. The **sysdbclose** routine is similarly invoked when the user disconnects from the database. (For more information about these session configuration routines, see the *HCL® Informix® Guide to SQL: Syntax* and the *HCL® Informix® Administrator's Guide*.)

An *external routine* is a routine written in C or some other external language.

The EXECUTE statements

You can use EXECUTE PROCEDURE or EXECUTE FUNCTION to execute an SPL routine or external routine. In general, it is best to use EXECUTE PROCEDURE with procedures and EXECUTE FUNCTION with functions.

Tip: For backward compatibility, the EXECUTE PROCEDURE statement allows you to use an SPL function name and an INTO clause to return values. However, *it is* recommended that you use EXECUTE PROCEDURE only with procedures and EXECUTE FUNCTION only with functions.

You can issue EXECUTE PROCEDURE and EXECUTE FUNCTION statements as stand-alone statements from or from within an SPL routine or external routine. If the routine name is unique within the database, and if it does not require arguments, you can execute it by entering just its name and parentheses after EXECUTE PROCEDURE, as the following figure shows.

Figure 481. Execute a procedure.

```
EXECUTE PROCEDURE update_orders();
```

The INTO clause is never present when you invoke a procedure with the EXECUTE statement because a procedure does not return a value.

If the routine expects arguments, you must enter the argument values within parentheses, as the following figure shows.

Figure 482. Execute a procedure with arguments.

```
EXECUTE FUNCTION scale_rectangles(107, 1.9)
   INTO new;
```

The statement executes a function. Because a function returns a value, EXECUTE FUNCTION uses an INTO clause that specifies a variable where the return value is stored. The INTO clause must always be present when you use an EXECUTE statement to execute a function.

If the database has more than one procedure or function of the same name, HCL Informix® locates the right function based on the data types of the arguments. For example, the statement in the previous figure supplies INTEGER and REAL values as arguments, so if your database contains multiple routines named scale_rectangles(), the database server executes only the scale_rectangles() function that accepts INTEGER and REAL data types.

The parameter list of an SPL routine always has parameter names as well as data types. When you execute the routine, the parameter names are optional. However, if you pass arguments by name (instead of just by value) to EXECUTE PROCEDURE or EXECUTE FUNCTION, as in the following figure, Informix® resolves the routine-by-routine name and arguments only, a process known as *partial routine resolution*.

Figure 483. Execute a routine passing arguments by name.

```
EXECUTE FUNCTION scale_rectangles( rectid = 107,
   scale = 1.9 ) INTO new_rectangle;
```

You can also execute an SPL routine stored on another database server by adding a *qualified routine name* to the statement; that is, a name in the form `database@dbserver:owner_name.routine_name`, as in the following figure.

Figure 484. Execute an SPL routine stored on another database server.

```
EXECUTE PROCEDURE informix@davinci:bsmith.update_orders();
```

When you execute a routine remotely, the `owner_name` in the qualified routine name is optional.

The CALL statement

You can call an SPL routine or an external routine from an SPL routine using the CALL statement. CALL can execute both procedures and functions. If you use CALL to execute a function, add a RETURNING clause and the name of an SPL variable (or variables) that will receive the value (or values) the function returns.

Suppose, for example, that you want the scale_rectangles function to call an external function that calculates the area of the rectangle and then returns the area with the rectangle description, as in the following figure.

Figure 485. Call an external function.

```
CREATE FUNCTION scale_rectangles( rectid INTEGER,
      scale REAL )
   RETURNING rectangle_t, REAL;

   DEFINE rectv rectangle_t;
   DEFINE a REAL;
   SELECT rect INTO rectv
      FROM rectangles WHERE id = rectid;
   IF ( rectv IS NULL ) THEN
      LET rectv.start = (0.0,0.0);
      LET rectv.length = 1.0;
      LET rectv.width = 1.0;
      LET a = 1.0;
      RETURN rectv, a;
   ELSE
      LET rectv.length = scale * rectv.length;
      LET rectv.width = scale * rectv.width;
      CALL area(rectv.length, rectv.width) RETURNING a;
      RETURN rectv, a;
   END IF;

END FUNCTION;
```

The SPL function uses a CALL statement that executes the external function area(). The value area() returns is stored in **a** and returned to the calling routine by the RETURN statement.

In this example, area() is an external function, but you can use CALL in the same manner with an SPL function.

Execute routines in expressions

Just as with built-in functions, you can execute SPL routines (and external routines from SPL routines) by using them in expressions in SQL and SPL statements. A routine used in an expression is usually a function, because it returns a value to the rest of the statement.

For example, you might execute a function by a LET statement that assigns the return value to a variable. The statements in the following figure perform the same task. They execute an external function within an SPL routine and assign the return value to the variable **a**.

Figure 486. Execute an external function within an SPL routine.

```
LET a = area( rectv.length, rectv.width );

CALL area( rectv.length, rectv.width ) RETURNING a;
   -- these statements are equivalent
```

You can also execute an SPL routine from an SQL statement, as the following figure shows. Suppose you write an SPL function, increase_by_pct, which increases a given price by a given percentage. After you write an SPL routine, it is available for use in any other SPL routine.

Figure 487. Execute an SPL routine from an SQL statement.

```
CREATE FUNCTION raise_price ( num INT )
   RETURNING DECIMAL;

   DEFINE p DECIMAL;

   SELECT increase_by_pct(price, 20) INTO p
      FROM inventory WHERE prod_num = num;

   RETURN p;

END FUNCTION;
```

The example selects the **price** column of a specified row of **inventory** and uses the value as an argument to the SPL function increase_by_pct. The function then returns the new value of **price**, increased by 20 percent, in the variable **p**.

Execute an external function with the RETURN statement

You can use a RETURN statement to execute any external function from within an SPL routine. The following figure shows an external function that is used in the RETURN statement of an SPL program.

Figure 488. A RETURN statement to execute an external function from within an SPL routine.

```
CREATE FUNCTION c_func() RETURNS int
LANGUAGE C;

CREATE FUNCTION spl_func() RETURNS INT;
   RETURN(c_func());
END FUNCTION;

EXECUTE FUNCTION spl_func();
```

When you execute the spl_func() function, the c_func() function is invoked, and the SPL function returns the value that the external function returns.

Execute cursor functions from an SPL routine

A cursor function is a user-defined function that returns one or more rows of data and therefore requires a cursor to execute. A cursor function can be either of the following functions:

- An SPL function whose RETURN statement includes WITH RESUME
- An external function that is defined as an iterator function

The behavior of a cursor function is the same whether the function is an SPL function or an external function. However, an SPL cursor function can return more than one value per iteration, whereas an external cursor function (iterator function) can return only one value per iteration.

To execute a cursor function from an SPL routine, you must include the function in a FOREACH loop of an SPL routine. The following examples show different ways to execute a cursor function in a FOREACH loop:

```
FOREACH SELECT cur_func1(col_name) INTO spl_var FROM tab1
   INSERT INTO tab2 VALUES (spl_var);
END FOREACH

FOREACH EXECUTE FUNCTION cur_func2() INTO spl_var
   INSERT INTO tab2 VALUES (spl_var);
END FOREACH
```

Dynamic routine-name specification

Dynamic routine-name specification allows you to execute an SPL routine from another SPL routine, by building the name of the called routine within the calling routine. Dynamic routine-name specification simplifies how you can write an SPL routine that calls another SPL routine whose name is not known until runtime. The database server lets you specify an SPL variable instead of the explicit name of an SPL routine in the EXECUTE PROCEDURE or EXECUTE FUNCTION statement.

In the following figure, the SPL procedure **company_proc** updates a large company sales table and then assigns an SPL variable named **salesperson_proc** to hold the dynamically created name of an SPL procedure that updates another, smaller table that contains the monthly sales of an individual salesperson.

Figure 489. Dynamic routine-name specification.

```
CREATE PROCEDURE company_proc ( no_of_items INT,
   itm_quantity SMALLINT, sale_amount MONEY,
   customer VARCHAR(50), sales_person VARCHAR(30) )

DEFINE salesperson_proc VARCHAR(60);

-- Update the company table
INSERT INTO company_tbl VALUES (no_of_items, itm_quantity,
   sale_amount, customer, sales_person);

-- Generate the procedure name for the variable salesperson_proc
LET salesperson_proc = sales_person || "." || "tbl" ||
   current_month || "_" || current_year || "_proc" ;

-- Execute the SPL procedure that the salesperson_proc
-- variable specifies
EXECUTE PROCEDURE salesperson_proc (no_of_items,
   itm_quantity, sale_amount, customer)
END PROCEDURE;
```

In example, the procedure **company_proc** accepts five arguments and inserts them into **company_tbl**. Then the LET statement uses various values and the concatenation operator || to generate the name of another SPL procedure to execute. In the LET statement:

sales_person

An argument passed to the **company_proc** procedure.

current_month

The current month in the system date.

current_year

The current year in the system date.

Therefore, if a salesperson named Bill makes a sale in July 1998, **company_proc** inserts a record in **company_tbl** and executes the SPL procedure **bill.tbl07_1998_proc**, which updates a smaller table that contains the monthly sales of an individual salesperson.

Rules for dynamic routine-name specification

You must define the SPL variable that holds the name of the dynamically executed SPL routine as CHAR, VARCHAR, NCHAR, or NVARCHAR type. You must also give the SPL variable a valid and non-NULL name.

The SPL routine that the dynamic routine-name specification identifies must exist before it can be executed. If you assign the SPL variable the name of a valid SPL routine, the EXECUTE PROCEDURE or EXECUTE FUNCTION statement executes the routine whose name is contained in the variable, even if a built-in function of the same name exists.

In an EXECUTE PROCEDURE or EXECUTE FUNCTION statement, you cannot use two SPL variables to create a variable name in the form *owner.routine_name*. However, you can use an SPL variable that contains a fully qualified routine name, for example, *bill.proc1*. The following figure shows both cases.

Figure 490. SPL variable that contains a fully qualified routine name.

```
EXECUTE PROCEDURE owner_variable.proc_variable;
   -- this is not allowed

LET proc1 = bill.proc1;
EXECUTE PROCEDURE proc1; -- this is allowed
```

Privileges on routines

Privileges differentiate users who can create a routine from users who can execute a routine. Some privileges accrue as part of other privileges. For example, the DBA privilege includes permissions to create routines, execute routines, and grant these privileges to other users.

Privileges for registering a routine

To register a routine in the database, an authorized user wraps the SPL commands in a CREATE FUNCTION or CREATE PROCEDURE statement. The database server stores a registered SPL routine internally. The following users qualify to register a new routine in the database:

- Any user with the DBA privilege can register a routine with or without the DBA keyword in the CREATE statement.

  For an explanation of the DBA keyword, see DBA privileges for executing a routine on page 1813.
- A user who does not have the DBA privilege needs the Resource privilege to register an SPL routine. The creator is the owner of the routine.

  A user who does not have the DBA privilege cannot use the DBA keyword to register the routine.

  A DBA must give other users the Resource privilege needed to create routines. The DBA can also revoke the Resource privilege, preventing the user from creating further routines.
- Besides holding the DBA privilege or the Resource privilege on the database in which the UDR is registered, the user who creates a UDR must also hold the Usage privilege on the programming language in which the UDR is written. These SQL statements can grant language-level Usage privileges for specific programming languages:
  - `GRANT USAGE ON LANGUAGE C`
  - `GRANT USAGE ON LANGUAGE JAVA`
  - `GRANT USAGE ON LANGUAGE SPL`

  Besides an individual user, the grantee of these privileges can also be a user-defined role, or the PUBLIC group. After language-level Usage privileges are granted to a role, any user who holds that role can enable all the access privileges of the role by using the SET ROLE statement of SQL to specify that role as the current role.

For external routines written in the C language or the Java™ language, if the IFX_EXTEND_ROLE configuration parameter is enabled, only users to whom the DBSA has granted EXTERNAL role has been granted can register, drop, or alter external UDRs or modules. This parameter is enabled by default. By setting the IFX_EXTEND_ROLE configuration parameter to `OFF` or to `0`, the DBSA can disable the requirement of holding the EXTEND role for DDL operations on modules and external UDRs. This security feature has no effect, however, on SPL routines.

In summary, a user who holds the database-level and language-level discretionary access privileges that are identified above (and who also holds the EXTEND role, if IFX_EXTEND_ROLE is enabled and the UDR is an external routine) can reference UDRs in the following SQL statements:

- The DBA or a user can register a new UDR with the CREATE FUNCTION, CREATE FUNCTION FROM, CREATE PROCEDURE, CREATE PROCEDURE FROM, CREATE ROUTINE, or CREATE ROUTINE FROM statement.
- The DBA or the owner of an existing UDR can cancel the registration of that UDR with the DROP FUNCTION, DROP PROCEDURE, or DROP ROUTINE statement.
- The DBA or the owner of an existing UDR can modify the definition of that UDR with the ALTER FUNCTION, ALTER PROCEDURE, or ALTER ROUTINE statement.

Privileges for executing a routine

The Execute privilege enables users to invoke a routine. The routine might be invoked by the EXECUTE or CALL statements, or by using a function in an expression. The following users have a default Execute privilege, which enables them to invoke a routine:

- By default, any user with the DBA privilege can execute any routine in the database.
- If the routine is registered with the qualified CREATE DBA FUNCTION or CREATE DBA PROCEDURE statements, only users with the DBA privilege have a default Execute privilege for that routine.
- If the database is not ANSI compliant, user **public** (any user with Connect database privilege) automatically has the Execute privilege to a routine that is not registered with the DBA keyword.
- In an ANSI-compliant database, the procedure owner and any user with the DBA privilege can execute the routine without receiving additional privileges.

Grant and revoke the Execute privilege

Routines have the following GRANT and REVOKE requirements:

- The DBA can grant or revoke the Execute privilege to any routine in the database.
- The creator of a routine can grant or revoke the Execute privilege on that particular routine. The creator forfeits the ability to grant or revoke by including the AS *grantor* clause with the GRANT EXECUTE ON statement.
- Another user can grant the Execute privilege if the owner applied the WITH GRANT keywords in the GRANT EXECUTE ON statement.

A DBA or the routine owner must explicitly grant the Execute privilege to non-DBA users for the following conditions:

- A routine that was registered with the DBA keyword
- A routine in an ANSI-compliant database
- A routine in a database that is not ANSI-compliant, but with the **NODEFDAC** environment variable set to `yes`.
-

An owner can restrict the Execute privilege on a routine even though the database server grants that privilege to public by default. To do this, issue the REVOKE EXECUTE ON PUBLIC statement. The DBA and owner can still execute the routine and can grant the Execute privilege to specific users, if applicable.

Execute privileges with COMMUTATOR and NEGATOR functions



Important: If you explicitly grant the Execute privilege on an SPL function that is the commutator or negator function of a UDR, you must also grant that privilege on the commutator or the negator function before the grantee can use either. You cannot specify COMMUTATOR or NEGATOR modifiers with SPL procedures.

The following example demonstrates both limiting privileges for a function and its negator to one group of users. Suppose you create the following pair of negator functions:

```
CREATE FUNCTION greater(y PERCENT, z PERCENT)
RETURNS BOOLEAN
NEGATOR= less(y PERCENT, z PERCENT);
. . .
CREATE FUNCTION less(y PERCENT, z PERCENT)
RETURNS BOOLEAN
NEGATOR= greater(y PERCENT, z PERCENT);
```

By default, any user can execute both the function and negator. The following statements allow only **accounting** to execute these functions:

```
REVOKE EXECUTE ON FUNCTION greater FROM PUBLIC;
REVOKE EXECUTE ON FUNCTION less FROM PUBLIC;
GRANT accounting TO mary, jim, ted;
GRANT EXECUTE ON FUNCTION greater TO accounting;
GRANT EXECUTE ON FUNCTION less TO accounting;
```

A user might receive the Execute privilege accompanied by the WITH GRANT OPTION authority to grant the Execute privilege to other users. If a user loses the Execute privilege on a routine, the Execute privilege is also revoked from all users who were granted the Execute privilege by that user.

For more information, see the GRANT and REVOKE statement descriptions in the *HCL® Informix® Guide to SQL: Syntax*.

Privileges on objects associated with a routine

The database server checks the existence of any referenced objects and verifies that the user invoking the routine has the necessary privileges to access the referenced objects.

Objects referenced by a routine can include:

- Tables and columns
- Sequence objects
- User-defined data types
- Other routines executed by the routine

When a routine is run, the effective privilege is defined as the union of:

- The privileges of the user running the routine,
- The privileges that the owner has with the GRANT option.

By default, the database administrator has all the privileges in a database with the GRANT option. Therefore, users executing routines that are owned by database administrators can select from all of the tables in a given database.

A GRANT EXECUTE ON statement confers to the grantee any table-level privileges that the grantor received from a GRANT statement that contained the WITH GRANT keywords.

The owner of the routine, and not the user who runs the routine, owns the unqualified objects created in the course of executing the routine. For example, assume user **howie** registers an SPL routine that creates two tables, with the following SPL routine:

```
CREATE PROCEDURE promo()
. . .
   CREATE TABLE newcatalog
   (
   catlog_num INTEGER
   cat_advert VARCHAR(255, 65)
   cat_picture BLOB
   ) ;
   CREATE TABLE dawn.mailers
   (
   cust_num INTEGER
   interested_in SET(catlog_num INTEGER)
   );
END PROCEDURE;
```

User **julia** runs the routine, which creates the table **newcatalog**. Because no owner name qualifies table name **newcatalog**, the routine owner (**howie**) owns **newcatalog**. By contrast, the qualified name **dawn.maillist** identifies **dawn** as the owner of **maillist**.

DBA privileges for executing a routine

If a DBA creates a routine using the DBA keyword, the database server automatically grants the Execute privilege only to other users with the DBA privilege. A DBA can, however, explicitly grant the Execute privilege on a DBA routine to a user who does not have the DBA privilege.

When a user executes a routine that was registered with the DBA keyword, that user assumes the privileges of a DBA for the duration of the routine. If a user who does not have the DBA privilege runs a DBA routine, the database server implicitly grants a temporary DBA privilege to the invoker. Before exiting a DBA routine, the database server implicitly revokes the temporary DBA privilege.

Objects created in the course of running a DBA routine are owned by the user who executes the routine, unless a statement in the routine explicitly names someone else as the owner. For example, suppose that **tony** registers the **promo()** routine with the DBA keyword, as follows:

```
CREATE DBA PROCEDURE promo()
   . . .
   CREATE TABLE catalog
   . . .
   CREATE TABLE libby.mailers
   . . .
END PROCEDURE;
```

Although **tony** owns the routine, if **marty** runs it, then **marty** owns the **catalog** table, but user **libby** owns **libby.mailers** because their name qualifies the table name, making them the table owner.

A called routine does not inherit the DBA privilege. If a DBA routine executes a routine that was created without the DBA keyword, the DBA privileges do not affect the called routine.

If a routine that is registered without the DBA keyword calls a DBA routine, the caller must have Execute privileges on the called DBA routine. Statements within the DBA routine execute as they would within any DBA routine.

The following example demonstrates what occurs when a DBA and non-DBA routine interact. Suppose procedure dbspc_cleanup() executes another procedure clust_catalog(). Suppose also that the procedure clust_catalog() creates an index and that the SPL source code for clust_catalog() includes the following statements:

```
CREATE CLUSTER INDEX c_clust_ix ON catalog (catalog_num);
```

The DBA procedure dbspc_cleanup() invokes the other routine with the following statement:

```
EXECUTE PROCEDURE clust_catalog(catalog);
```

Assume **tony** registered dbspc_cleanup() as a DBA procedure and clust_catalog() is registered without the DBA keyword, as the following statements show:

```
CREATE DBA PROCEDURE dbspc_cleanup(loc CHAR)
CREATE PROCEDURE clust_catalog(catalog CHAR)
GRANT EXECUTE ON dbspc_cleanup(CHAR) to marty;
```

Suppose user **marty** runs dbspc_cleanup(). Because index **c_clust_ix** is created by a non-DBA routine, **tony**, who owns both routines, also owns **c_clust_ix**. By contrast, **marty** would own index **c_clust_ix** if clust_catalog() is a DBA procedure, as the following registering and grant statements show:

```
CREATE PROCEDURE dbspc_cleanup(loc CHAR);
CREATE DBA PROCEDURE clust_catalog(catalog CHAR);
GRANT EXECUTE ON clust_catalog(CHAR) to marty;
```

Notice that dbspc_cleanup() need not be a DBA procedure to call a DBA procedure.

Find errors in an SPL routine

When you use CREATE PROCEDURE or CREATE FUNCTION to write an SPL routine with , the statement fails when you select **Run** from the menu, if a syntax error occurs in the body of the routine.

If you are creating the routine in , when you choose the **Modify** option from the menu, the cursor moves to the line that contains the syntax error. You can select **Run** and **Modify** again to check subsequent lines.

Compile-time warnings

If the database server detects a potential problem, but the syntax of the SPL routine is correct, the database server generates a warning and places it in a listing file. You can examine this file to check for potential problems before you execute the routine.

The file name and path name of the listing file are specified in the WITH LISTING IN clause of the CREATE PROCEDURE or CREATE FUNCTION statement. For information about how to specify the path name of the listing file, see Specify a DOCUMENT clause on page 1761.

If you are working on a network, the listing file is created on the system where the database resides. If you provide an absolute path name and file name for the file, the file is created at the location you specify.

For UNIX™, if you provide a relative path name for the listing file, the file is created in your home directory on the computer where the database resides. (If you do not have a home directory, the file is created in the **root** directory.)

For Windows™, if you provide a relative path name for the listing file, the default directory is your current working directory if the database is on the local computer. Otherwise the default directory is `%INFORMIXDIR%\bin`.

After you create the routine, you can view the file that is specified in the WITH LISTING IN clause to see the warnings that it contains.

Generate the text of the routine

After you create an SPL routine, it is stored in the **sysprocbody** system catalog table. The **sysprocbody** system catalog table contains the executable routine, as well as its text.

To retrieve the text of the routine, select the **data** column from the **sysprocbody** system catalog table. The **datakey** column for a text entry has the code **T**.

The SELECT statement in the following figure reads the text of the SPL routine **read_address**.

Figure 491. SELECT statement to read the text of the SPL routine.

```
SELECT data FROM informix.sysprocbody
   WHERE datakey = 'T'                  -- find text lines
   AND procid =
      ( SELECT procid
      FROM informix.sysprocedures
      WHERE informix.sysprocedures.procname =
         'read_address' )
```

Debug an SPL routine

After you successfully create and run an SPL routine, you can encounter logic errors. If the routine has logic errors, use the TRACE statement to help find them. You can trace the values of the following items:

- Variables
- Arguments
- Return values
- SQL error codes
- error codes

To generate a listing of traced values, first use the SQL statement SET DEBUG FILE to name the file that is to contain the traced output. When you create the SPL routine, include a TRACE statement.

The following methods specify the form of TRACE output.

Statement

Action

TRACE ON

Traces all statements except SQL statements. Prints the contents of variables before they are used. Traces routine calls and returned values.

TRACE PROCEDURE

Traces only the routine calls and returned values.

TRACE *expression*

Prints a literal or an expression. If necessary, the value of the expression is calculated before it is sent to the file.

The following figure demonstrates how you can use the TRACE statement to monitor how an SPL function executes.

Figure 492. The TRACE statement.

```
CREATE FUNCTION read_many  (lastname CHAR(15))
  RETURNING CHAR(15), CHAR(15), CHAR(20), CHAR(15),
    CHAR(2), CHAR(5);

  DEFINE p_lname,p_fname, p_city CHAR(15);
  DEFINE p_add CHAR(20);
  DEFINE p_state CHAR(2);
  DEFINE p_zip CHAR(5);
  DEFINE lcount, i INT;

  LET lcount = 1;

  TRACE ON;      -- Trace every expression from here on
  TRACE 'Foreach starts';  -- Trace statement with a literal

  FOREACH
  SELECT fname, lname, address1, city, state, zipcode
    INTO p_fname, p_lname, p_add, p_city, p_state, p_zip

    FROM customer
        WHERE lname = lastname
  RETURN p_fname, p_lname, p_add, p_city, p_state, p_zip
    WITH RESUME;
  LET lcount = lcount + 1;   -- count of returned addresses
  END FOREACH

  TRACE 'Loop starts';       -- Another literal
  FOR i IN (1 TO 5)
    BEGIN
      RETURN i , i+1, i*i, i/i, i-1,i WITH RESUME;
    END
  END FOR;

END FUNCTION;
```

With the TRACE ON statement, each time you execute the traced routine, entries are added to the file you specified in the SET DEBUG FILE statement. To see the debug entries, view the output file with any text editor.

The following list contains some of the output that the function in previous example generates. Next to each traced statement is an explanation of its contents.

Statement

Action

TRACE ON

Echoes TRACE ON statement.

TRACE Foreach starts

Traces expression, in this case, the literal string Foreach starts.

start select cursor

Provides notification that a cursor is opened to handle a FOREACH loop.

select cursor iteration

Provides notification of the start of each iteration of the select cursor.

expression: (+lcount, 1)

Evaluates the encountered expression, (lcount+1), to 2.

let lcount = 2

Echoes each LET statement with the value.

Exception handling

You can use the ON EXCEPTION statement to trap any exception (or error) that the database server returns to your SPL routine or any exception that the routine raises. The RAISE EXCEPTION statement lets you generate an exception within the SPL routine.

In an SPL routine, you cannot use exception handling to handle the following conditions:

- Success (row returned)
- Success (no rows returned)

Error trapping and recovering

The ON EXCEPTION statement provides a mechanism to trap any error.

To trap an error, enclose a group of statements in a statement block marked with BEGIN and END and add an ON EXCEPTION IN statement at the beginning of the statement block. If an error occurs in the block that follows the ON EXCEPTION statement, you can take recovery action.

The following figure shows an ON EXCEPTION statement within a statement block.

Figure 493. Trap errors.

```
BEGIN
DEFINE c INT;
ON EXCEPTION IN
  (
  -206, -- table does not exist
  -217  -- column does not exist
  ) SET err_num

IF err_num = -206 THEN
    CREATE TABLE t (c INT);
    INSERT INTO t VALUES (10);
    -- continue after the insert statement
  ELSE
    ALTER TABLE t ADD(d INT);
    LET c = (SELECT d FROM t);
    -- continue after the select statement.
  END IF
END EXCEPTION WITH RESUME

INSERT INTO t VALUES (10);  -- fails if t does not exist

LET c = (SELECT d FROM t);  -- fails if d does not exist
END
```

When an error occurs, the SPL interpreter searches for the innermost ON EXCEPTION declaration that traps the error. The first action after trapping the error is to reset the error. When execution of the error action code is complete, and if the ON EXCEPTION declaration that was raised included the WITH RESUME keywords, execution resumes automatically with the statement *following* the statement that generated the error. If the ON EXCEPTION declaration did not include the WITH RESUME keywords, execution exits the current block entirely.

Scope of control of an ON EXCEPTION statement

The scope of the ON EXCEPTION statement extends from the statement that immediately follows the ON EXCEPTION statement, and ends at the end of the statement block in which the ON EXCEPTION statement is issued. If the SPL routine includes no explicit statement blocks, the scope is all subsequent statements in the routine.

For the exceptions specified in the IN clause (or for all exceptions, if no IN clause is specified), the scope of the ON EXCEPTION statement includes all statements that follow the ON EXCEPTION statement within the same statement block. If other statement blocks are nested within that block, the scope also includes all statements in the nested statement blocks that follow the ON EXCEPTION statement, and any statements in statement blocks that are nested within those nested blocks.

The following pseudocode shows where the exception is valid within the routine. That is, if error `201` occurs in any of the indicated blocks, the action labeled *a201* occurs.

Figure 494. ON EXCEPTION statement scope of control.

```
CREATE PROCEDURE scope()
  DEFINE i INT;
       . . .
  BEGIN          -- begin statement block A
       . . .
    ON EXCEPTION IN (201)
    -- do action a201
    END EXCEPTION
    BEGIN         -- nested statement block aa
      -- do action, a201 valid here
    END
    BEGIN         -- nested statement block bb
      -- do action, a201 valid here
    END
    WHILE i < 10
      -- do something, a201 is valid here
    END WHILE

  END    -- end of statement block A
  BEGIN           -- begin statement block B
    -- do something
    -- a201 is NOT valid here
  END
END PROCEDURE;
```

User-generated exceptions

You can generate your own error using the RAISE EXCEPTION statement, as the following figure shows.

Figure 495. The RAISE EXCEPTION statement.

```
BEGIN
  ON EXCEPTION SET esql, eisam   -- trap all errors
    IF esql = -206 THEN          -- table not found
      -- recover somehow
    ELSE
      RAISE exception esql, eisam;  -- pass the error up
    END IF
  END EXCEPTION
    -- do something
END
```

In the example, the ON EXCEPTION statement uses two variables, **esql** and **eisam**, to hold the error numbers that the database server returns. The IF clause executes if an error occurs and if the SQL error number is -206. If any other SQL error is caught, it is passed out of this BEGINEND block to the last BEGINEND block of the previous example.

Simulate SQL errors

You can generate errors to simulate SQL errors, as the following figure shows. If the user is **pault**, then the SPL routine acts as if that user has no update privileges, even if the user really does have that privilege.

Figure 496. Simulate SQL errors.

```
BEGIN
  IF user = 'pault' THEN
     RAISE EXCEPTION -273;  -- deny Paul update privilege
  END IF
END
```

RAISE EXCEPTION to exit nested code

The following figure shows how you can use the RAISE EXCEPTION statement to break out of a deeply nested block.

Figure 497. The RAISE EXCEPTION statement.

```
BEGIN
  ON EXCEPTION IN (1)
  END EXCEPTION WITH RESUME -- do nothing significant (cont)

  BEGIN
    FOR i IN (1 TO 1000)
      FOREACH select ..INTO aa FROM t
        IF aa < 0 THEN
          RAISE EXCEPTION 1;     -- emergency exit
        END IF
      END FOREACH
    END FOR
    RETURN 1;
  END

  --do something;               -- emergency exit to
                                 -- this statement.
  TRACE 'Negative value returned';
  RETURN -10;
END
```

If the innermost condition is true (if **aa** is negative), then the exception is raised and execution jumps to the code following the END of the block. In this case, execution jumps to the TRACE statement.

Remember that a BEGINEND block is a *single* statement. If an error occurs somewhere inside a block and the trap is outside the block, the rest of the block is skipped when execution resumes, and execution begins at the next statement.

Unless you set a trap for this error somewhere in the block, the error condition is passed back to the block that contains the call and back to any blocks that contain the block. If no ON EXCEPTION statement exists that is set to handle the error, execution of the SPL routine stops, creating an error for the routine that is executing the SPL routine.

Check the number of rows processed in an SPL routine

Within SPL routines, you can use the DBINFO function to find out the number of rows that have been processed in SELECT, INSERT, UPDATE, DELETE, EXECUTE PROCEDURE, and EXECUTE FUNCTION statements.

The following figure shows an SPL function that uses the DBINFO function with the 'sqlca.sqlerrd2' option to determine the number of rows that are deleted from a table.

Figure 498. Determine the number of rows deleted from a table.

```
CREATE FUNCTION del_rows ( pnumb INT )
RETURNING INT;

DEFINE nrows INT;

DELETE FROM sec_tab WHERE part_num = pnumb;
LET nrows = DBINFO('sqlca.sqlerrd2');

RETURN nrows;

END FUNCTION;
```

To ensure valid results, use this option after SELECT and EXECUTE PROCEDURE or EXECUTE FUNCTION statements have completed executing. In addition, if you use the 'sqlca.sqlerrd2' option within cursors, make sure that all rows are fetched before the cursors are closed, to ensure valid results.

Summary

SPL routines provide many opportunities for streamlining your database process, including enhanced database performance, simplified applications, and limited or monitored access to data. You can also use SPL routines to handle extended data types, such as collection types, row types, opaque types, and distinct types. For syntax diagrams of SPL statements, see the *HCL® Informix® Guide to SQL: Syntax*.

Create and use triggers

This section describes each component of the CREATE TRIGGER statement, illustrates some uses for triggers, and describes the advantages of using an SPL routine as a triggered action.

In addition, this section describes INSTEAD OF trigger that can be defined on views.

An SQL trigger is a mechanism that resides in the database. It is available to any user who has permission to use it. An SQL trigger specifies that when a data-manipulation language (DML) operation (an INSERT, SELECT, DELETE, or UPDATE statement) occurs on a particular table, the database server automatically performs one or more additional actions. For triggers defined on views, the triggered action on the base tables of the view replaces the triggering event. For triggers on tables or views, the triggered actions can be INSERT, DELETE, UPDATE, EXECUTE PROCEDURE or EXECUTE FUNCTION statements.

HCL Informix® also supports user-defined routines written in C or in Java™ as triggered actions.

For information on how to write a C UDR to obtain metadata information about trigger events, see the *HCL® Informix® DataBlade® API Programmer's Guide*.

When to use triggers

Because a trigger resides in the database and anyone who has the required privilege can use it, a trigger lets you write a set of SQL statements that multiple applications can use. It lets you avoid redundant code when multiple programs need to perform the same database operation.

You can use triggers to perform the following actions, as well as others that are not found in this list:

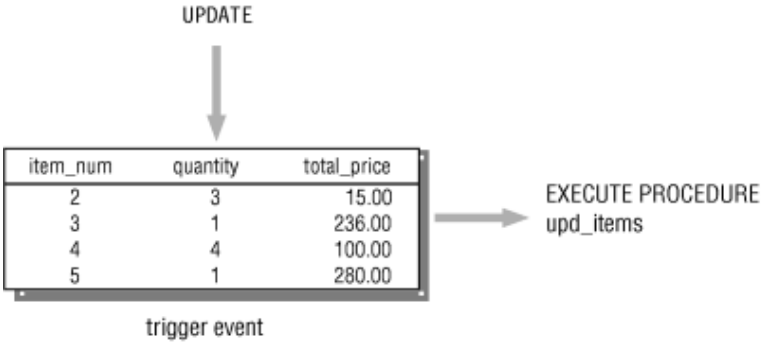
- Create an audit trail of activity in the database. For example, you can track updates to the orders table by updating corroborating information to an audit table.
- Implement a business rule. For example, you can determine when an order exceeds a customer's credit limit and display a message to that effect.
- Derive additional data that is not available within a table or within the database. For example, when an update occurs to the **quantity** column of the **items** table, you can calculate the corresponding adjustment to the **total_price** column.
- Enforce referential integrity. When you delete a customer, for example, you can use a trigger to delete corresponding rows that have the same customer number in the **orders** table.

How to create a trigger

You use the CREATE TRIGGER statement to define a new trigger. The CREATE TRIGGER statement is a data-definition statement that associates SQL statements, called the *triggered action*, with a precipitating event on a table. When the event occurs, it triggers the associated SQL statements, which are stored in the database.

In this example, the triggering event is an UPDATE statement that references the **quantity** column of the **items** table. The following figure illustrates the relationship of the DML operation that activates the trigger, called the trigger event, to the triggered action.

Figure 499. Trigger event and triggered action



The CREATE TRIGGER statement consists of clauses that perform the following actions:

- Declare a name for the trigger .
- Specify the DML operation on a specified table or view as the triggering event.
- Define the SQL operations that this event triggers.

An optional clause, called the REFERENCING clause, is discussed in FOR EACH ROW triggered actions on page 1825.

To create a trigger, use or one of the SQL APIs. This section describes the CREATE TRIGGER statement as you enter it with the interactive Query-language option in . In an SQL API, you precede the statement with the symbol or keywords that identify it as an embedded statement.

Declare a trigger name

The trigger name identifies the trigger, and must be unique among trigger names within the database. The trigger name follows the words CREATE TRIGGER in the statement. Like any SQL identifier, can be up to 128 bytes in length, beginning with a letter and consisting of letters, digits, and the underscore (_) symbol. In the following example, the portion of the CREATE TRIGGER statement that is shown declares the name **upqty** for the trigger:

```
CREATE TRIGGER upqty       -- declare trigger name
```

Specify the trigger event

The *trigger event* is the type of DML statement that activates the trigger. When a statement of this type is performed on the table, the database server executes the SQL statements that make up the triggered action. For tables, the trigger event can be an INSERT, SELECT, DELETE, or UPDATE statement. For UPDATE or SELECT trigger event, you can specify one or more columns in the table to activate the trigger. If you do not specify any columns, then an UPDATE or SELECT of any column in the table activates the trigger. You can define multiple INSERT, DELETE, UPDATE and SELECT triggers on the same table, and multiple INSERT, DELETE, and UPDATE triggers on the same view.

You can only create a trigger on a table or view in the current database. Triggers cannot reference a remote table or view.

In the following excerpt from a CREATE TRIGGER statement, the trigger event is defined as an update of the **quantity** column in the **items** table:

```
CREATE TRIGGER upqty
   UPDATE OF quantity ON items     -- an UPDATE trigger event
```

This portion of the statement identifies the table on which you define the trigger. If the trigger event is an insert or delete operation, only the type of statement and the table name are required, as the following example shows:

```
CREATE TRIGGER ins_qty
   INSERT ON items                  -- an INSERT trigger event
```

Define the triggered actions

The *triggered actions* are the SQL statements that are performed when the trigger event occurs. The triggered actions can consist of INSERT, DELETE, UPDATE, EXECUTE FUNCTION and EXECUTE PROCEDURE statements. In addition to specifying what actions are to be performed, however, you must also specify when they are to be performed in relation to the triggering statement. You have the following choices:

- Before the triggering statement executes
- After the triggering statement executes
- For each row that is affected by the triggering statement

A single trigger on a table can define actions for each of these times.

To define a triggered action, specify when it occurs and then provide the SQL statement or statements to execute. You specify when the action is to occur with the keywords BEFORE, AFTER, or FOR EACH ROW. The triggered actions follow, enclosed in parentheses. The following triggered-action definition specifies that the SPL routine upd_items_p1 is to be executed before the triggering statement:

```
BEFORE(EXECUTE PROCEDURE upd_items_p1) -- a BEFORE action
```

A complete CREATE TRIGGER statement

To define a complete CREATE TRIGGER statement, combine the trigger-name clause, the trigger-event clause, and the triggered-action clause. The following CREATE TRIGGER statement is the result of combining the components of the statement from the preceding examples. This trigger executes the SPL routine upd_items_p1 whenever the **quantity** column of the **items** table is updated.

```
CREATE TRIGGER upqty
   UPDATE OF quantity ON items
   BEFORE(EXECUTE PROCEDURE upd_items_p1);
```

If a database object in the trigger definition, such as the SPL routine upd_items_p1 in this example, does not exist when the database server processes the CREATE TRIGGER statement, it returns an error.

Triggered actions

To use triggers effectively, you need to understand the relationship between the triggering statement and the resulting triggered actions. You define this relationship when you specify the time that the triggered action occurs; that is, BEFORE, AFTER, or FOR EACH ROW.

BEFORE and AFTER triggered actions

Triggered actions that occur before or after the trigger event execute only once. A BEFORE triggered action executes before the *triggering statement*, that is, before the occurrence of the trigger event. An AFTER triggered action executes after the action of the triggering statement is complete. BEFORE and AFTER triggered actions execute even if the triggering statement does not process any rows.

Among other uses, you can use BEFORE and AFTER triggered actions to determine the effect of the triggering statement. For example, before you update the **quantity** column in the **items** table, you could call the SPL routine upd_items_p1 to calculate the total quantity on order for all items in the table, as the following example shows. The procedure stores the total in a global variable called **old_qty**.

```
CREATE PROCEDURE upd_items_p1()
   DEFINE GLOBAL old_qty INT DEFAULT 0;
   LET old_qty = (SELECT SUM(quantity) FROM items);
END PROCEDURE;
```

After the triggering update completes, you can calculate the total again to see how much it has changed. The following SPL routine, upd_items_p2, calculates the total of **quantity** again and stores the result in the local variable **new_qty**. Then it compares **new_qty** to the global variable **old_qty** to see if the total quantity for all orders has increased by more than 50 percent. If so, the procedure uses the RAISE EXCEPTION statement to simulate an SQL error.

```
CREATE PROCEDURE upd_items_p2()
   DEFINE GLOBAL old_qty INT DEFAULT 0;
   DEFINE new_qty INT;
   LET new_qty = (SELECT SUM(quantity) FROM items);
   IF new_qty > old_qty * 1.50 THEN
      RAISE EXCEPTION -746, 0, 'Not allowed - rule violation';
   END IF
END PROCEDURE;
```

The following trigger calls upd_items_p1 and upd_items_p2 to prevent an extraordinary update on the **quantity** column of the **items** table:

```
CREATE TRIGGER up_items
   UPDATE OF quantity ON items
      BEFORE(EXECUTE PROCEDURE upd_items_p1())
      AFTER(EXECUTE PROCEDURE upd_items_p2());
```

If an update raises the total quantity on order for all items by more than 50 percent, the RAISE EXCEPTION statement in upd_items_p2 terminates the trigger with an error. When a trigger fails in a database that has transaction logging, the database server rolls back the changes that both the triggering statement and the triggered actions make. For more information on what happens when a trigger fails, see the CREATE TRIGGER statement in the *HCL® Informix® Guide to SQL: Syntax*.

FOR EACH ROW triggered actions

A FOR EACH ROW triggered action executes once for each row that the triggering statement affects. For example, if the triggering statement has the following syntax, a FOR EACH ROW triggered action executes once for each row in the **items** table in which the **manu_code** column has a value of `'KAR'`:

```
UPDATE items SET quantity = quantity * 2
   WHERE manu_code = 'KAR';
```

If the triggering event does not process any rows, a FOR EACH ROW triggered action does not execute.

For a trigger on a table, if the triggering event is a SELECT statement, the trigger is a called a Select trigger, and the triggered actions execute after all processing on the retrieved row is complete. The triggered actions might not execute immediately; however, because a FOR EACH ROW action executes for every instance of a row that the query returns. For example, in a SELECT statement with an ORDER BY clause, all rows must be qualified against the WHERE clause before they are sorted and returned.

The REFERENCING clause

When you create a FOR EACH ROW triggered action, you must usually indicate in the triggered action statements whether you are referring to the value of a column before or after the effect of the triggering statement. For example, imagine that you want to track updates to the **quantity** column of the **items** table. To do this, create the following table to record the activity:

```
CREATE TABLE log_record
   (item_num     SMALLINT,
   ord_num       INTEGER,
   username      CHARACTER(8),
   update_time   DATETIME YEAR TO MINUTE,
```

```
   old_qty       SMALLINT,
   new_qty       SMALLINT);
```

To supply values for the **old_qty** and **new_qty** columns in this table, you must be able to refer to the old and new values of **quantity** in the **items** table; that is, the values before and after the effect of the triggering statement. The REFERENCING clause enables you to do this.

The REFERENCING clause lets you create two prefixes that you can combine with a column name, one to reference the old value of the column, and one to reference its new value. These prefixes are called *correlation names*. You can create one or both correlation names, depending on your requirements. You indicate which one you are creating with the keywords OLD and NEW. The following REFERENCING clause creates the correlation names **pre_upd** and **post_upd** to refer to the old and new values in a row:

```
REFERENCING OLD AS pre_upd NEW AS post_upd
```

The following triggered action creates a row in **log_record** when **quantity** is updated in a row of the **items** table. The INSERT statement refers to the old values of the **item_num** and **order_num** columns and to both the old and new values of the **quantity** column.

```
FOR EACH ROW(INSERT INTO log_record
  VALUES (pre_upd.item_num, pre_upd.order_num, USER,
          CURRENT, pre_upd.quantity, post_upd.quantity));
```

The correlation names defined in the REFERENCING clause apply to all rows that the triggering statement affects.

Important: If you refer to a column name that is not qualified by a correlation name, the database server makes no special effort to search for the column in the definition of the triggering table. You must always use a correlation name with a column name in SQL statements in a FOR EACH ROW triggered action, unless the statement is valid independent of the triggered action. For more information, see the CREATE TRIGGER statement in the *HCL® Informix® Guide to SQL: Syntax*.

The WHEN condition

As an option for triggers on tables, you can precede a triggered action with a WHEN clause to make the action dependent on the outcome of a test. The WHEN clause consists of the keyword WHEN followed by the condition statement given in parentheses. In the CREATE TRIGGER statement, the WHEN clause follows the keywords BEFORE, AFTER, or FOR EACH ROW and precedes the triggered-action list.

When a WHEN condition is present, if it evaluates to *true*, the triggered actions execute in the order in which they appear. If the WHEN condition evaluates to *false* or *unknown*, the actions in the triggered-action list do not execute. If the trigger specifies FOR EACH ROW, the condition is evaluated for each row also.

In the following trigger example, the triggered action executes only if the condition in the WHEN clause is true; that is, if the post-update unit price is greater than two times the pre-update unit price:

```
CREATE TRIGGER up_price
   UPDATE OF unit_price ON stock
   REFERENCING OLD AS pre NEW AS post
   FOR EACH ROW WHEN(post.unit_price > pre.unit_price * 2)
```

```
   (INSERT INTO warn_tab
    VALUES(pre.stock_num, pre.manu_code, pre.unit_price,
           post.unit_price, CURRENT));
```

For more information on the WHEN condition, see the CREATE TRIGGER statement in the *HCL® Informix® Guide to SQL: Syntax*.

SPL routines as triggered actions

Probably the most powerful feature of triggers is the ability to call an SPL routine as a triggered action. The EXECUTE PROCEDURE or EXECUTE FUNCTION statement, which calls an SPL routine, lets you pass data from the triggering table to the SPL routine and also to update the triggering table with data returned by the SPL routine. SPL also lets you define variables, assign data to them, make comparisons, and use procedural statements to accomplish complex tasks within a triggered action.

Pass data to an SPL routine

You can pass data to an SPL routine in the argument list of the EXECUTE PROCEDURE or EXECUTE FUNCTION statement. The EXECUTE PROCEDURE statement in the following example passes values from the **quantity** and **total_price** columns of the **items** table to the SPL routine **calc_totpr**:

```
CREATE TRIGGER upd_totpr
   UPDATE OF quantity ON items
   REFERENCING OLD AS pre_upd NEW AS post_upd
   FOR EACH ROW(EXECUTE PROCEDURE calc_totpr(pre_upd.quantity,
      post_upd.quantity, pre_upd.total_price) INTO total_price);
```

Passing data to an SPL routine lets you use data values in the operations that the routine performs.

Using SPL

The EXECUTE PROCEDURE statement in the preceding trigger calls the SPL routine that the following example shows. The procedure uses SPL to calculate the change that needs to be made to the **total_price** column when **quantity** is updated in the **items** table. The procedure receives both the old and new values of **quantity** and the old value of **total_price**. It divides the old total price by the old quantity to derive the unit price. It then multiplies the unit price by the new quantity to obtain the new total price.

```
CREATE PROCEDURE calc_totpr(old_qty SMALLINT, new_qty SMALLINT,
   total MONEY(8)) RETURNING MONEY(8);
   DEFINE u_price LIKE items.total_price;
   DEFINE n_total LIKE items.total_price;
   LET u_price = total / old_qty;
   LET n_total = new_qty * u_price;
   RETURN n_total;
END PROCEDURE;
```

In this example, SPL lets the trigger derive data that is not directly available from the triggering table.

Update nontriggering columns with data from an SPL routine

Within a triggered action, the INTO clause of the EXECUTE PROCEDURE statement lets you update nontriggering columns in the triggering table. The EXECUTE PROCEDURE statement in the following example calls the **calc_totpr** SPL procedure that contains an INTO clause, which references the column **total_price**:

```
FOR EACH ROW(EXECUTE PROCEDURE calc_totpr(pre_upd.quantity,
   post_upd.quantity, pre_upd.total_price) INTO total_price);
```

The value that is updated into **total_price** is returned by the RETURN statement at the conclusion of the SPL procedure. The **total_price** column is updated for each row that the triggering statement affects.

Trigger routines

You can define specialized SPL routines, called *trigger routines*, that can be invoked only from the FOR EACH ROW section of the triggered action. Unlike ordinary UDRs that EXECUTE FUNCTION or EXECUTE PROCEDURE routines can call from the triggered action list, trigger routines include their own REFERENCING clause that defines correlation names for the old and new column values in rows that the triggered action modifies. These correlation names can be referenced in SPL statements within the trigger routine, providing greater flexibility in how the triggered action can modify data in the table or view.

Trigger routines can also use trigger-type Boolean operators, called DELETING, INSERTING, SELECTING, and UPDATING, to identify what type of trigger has called the trigger routine. Trigger routines can also invoke the **mi_trigger*** routines, which are sometimes called *trigger introspection routines*, to obtain information about the context in which the trigger routine has been called.

Trigger routines are invoked by EXECUTE FUNCTION or EXECUTE PROCEDURE statements that include the WITH TRIGGER REFERENCES keywords. These statements must call the trigger routine from the FOR EACH ROW section of the triggered action, rather than from the BEFORE or AFTER sections.

For information about syntax features that the CREATE FUNCTION, CREATE PROCEDURE, EXECUTE FUNCTION, and EXECUTE PROCEDURE statements of SQL support for defining and executing trigger routines, see your *HCL® Informix® Guide to SQL: Syntax*. For more information about the **mi_trigger*** routines, see your *HCL® Informix® DataBlade® API Programmer's Guide*.

Triggers in a table hierarchy

When you define a trigger on a supertable, any subtables in the table hierarchy also inherit the trigger. Consequently when you perform operations on tables in the hierarchy, triggers can execute for any table in the hierarchy that is a subtable of the table on which a trigger is defined.

Select triggers

When the CREATE TRIGGER statement defines as its triggering event any query on a specific table (

```
SELECT ON table
```

or

```
SELECT ON column-list ON table
```

), the resulting trigger object is a *Select trigger* on the specified *table*. The same trigger can also be activated by queries on a view that includes triggering columns from *table* as its base table. SELECT statements cannot, however, be the trigger events for INSTEAD OF triggers on a view.

If the CREATE TRIGGER statement also includes a *column-list* in the definition of an enabled Select trigger event, and the Projection list of a subsequent query on the specified table does not include any of the specified columns, that query cannot be a triggering event for the Select trigger.

Warning:

Select triggers are not reliable for auditing. Do not attempt to create a Select trigger on a table, or on a subset of its columns, for the purpose of performing application-specific auditing. In general, it is not possible, to track the number of SELECT actions on a table by creating a Select trigger to insert an audit record into an audit table each time a user queries a certain table.

For example, suppose that you define a Select trigger on the table **`AuditedTable`** and that a user who holds Select privileges on **`AuditedTable`** issues the following query:

```
SELECT a.* FROM (SELECT * FROM AuditedTable) AS a;
```

The database server issues no error, but the SELECT trigger on **`AuditedTable`** will not be activated by this query. A query that included a set operator, such as UNION or INTERSECT, or any other syntax that Select triggers do not support, would be similarly invisible to an audit-record strategy that is based on Select triggers.

Because of the numerous restrictions on the execution of Select triggers, as partially listed in this chapter, the resulting Select trigger actions will typically correspond to only a subset (that might be empty) of whatever logical Select events you are attempting to enumerate.

SELECT statements that execute triggered actions

When you create a select trigger, only certain types of select statements can execute the actions defined on that trigger. A select trigger executes for the following types of SELECT statements only:

- Stand-alone SELECT statements
- Collection subqueries in the select list of a SELECT statement
- SELECT statements embedded in user-defined routines
- Views

Stand-alone SELECT statements

Suppose you define the following Select trigger on a table:

```
CREATE TRIGGER hits_trig SELECT OF col_a ON tab_a
   REFERENCING OLD AS hit
   FOR EACH ROW (INSERT INTO hits_log
                 VALUES (hit.col_a, CURRENT, USER));
```

A Select trigger executes when the triggering column appears in the select list of a stand-alone SELECT statement. The following statement executes a triggered action on the **hits_trig** trigger for each instance of a row that the database server returns:

```
SELECT col_a FROM tab_a;
```

Collection subqueries in the projection list of a query

A Select trigger executes when the triggering column appears in a collection subquery that occurs in the projection list of another SELECT statement. The following statement executes a triggered action on the **hits_trig** trigger for each instance of a row that the collection subquery returns:

```
SELECT MULTISET(SELECT col_a FROM tab_a) FROM ...
```

SELECT statements embedded in user-defined routines

A select trigger that is defined on a SELECT statement embedded in a user defined routine (UDR) executes a triggered action in the following instances only:

- The UDR appears in the select list of a SELECT statement
- The UDR is invoked with an EXECUTE PROCEDURE statement

Suppose you create a routine **new_proc** that contains the statement `SELECT col_a FROM tab_a`. Each of the following statements executes a triggered action on the **hits_trig** trigger for each instance of a row that the embedded SELECT statement returns:

```
SELECT new_proc() FROM tab_b;
EXECUTE PROCEDURE new_proc;
```

Views

Select triggers execute a triggered action for views whose base tables contain a reference to a triggering column. You cannot, however, define a Select trigger on a view.

Suppose you create the following view:

```
CREATE VIEW view_tab AS
   SELECT * FROM tab_a;
```

The following statements execute a triggered action on the **hits_trig** trigger for each instance of a row that the view returns:

```
SELECT * FROM view_tab;

SELECT col_a FROM tab_a;
```

Restrictions on execution of Select triggers

The following types of SELECT statements do not trigger any actions when they reference a table or column on which an enabled Select trigger is defined.

- No triggering column is referenced in the Projection list (for example, a column that appears only in the WHERE clause of a SELECT statement does not execute a Select trigger).
- The SELECT statement references a remote table.
- The SELECT statement calls an aggregate function or an OLAP window aggregation function.
- The SELECT statement includes a set operator (UNION, UNION ALL, INTERSECT, MINUS, or EXCEPT)
- The SELECT statement includes the DISTINCT or UNIQUE keyword.
- The UDR expression that contains the SELECT statement is not in the Projection list.
- The SELECT statement appears within an INSERT INTO statement.
- The SELECT statement appears within a scroll cursor.
- The trigger is a cascading Select trigger.

  A cascading Select trigger is a trigger whose actions includes an SPL routine that itself has a triggering SELECT statement. The actions of a cascading Select trigger do not execute, however, and the database server does not return an error.

Select triggers on tables in a table hierarchy

When you define a select trigger on a supertable, any subtables in the table hierarchy also inherit the trigger.

For information about overriding and disabling inherited triggers, see Triggers in a table hierarchy on page 1828.

Re-entrant triggers

A *re-entrant trigger* refers to a case in which the triggered action can reference the triggering table. In other words, both the triggering event and the triggered action can operate on the same table. For example, suppose the following UPDATE statement represents the triggering event:

```
UPDATE tab1 SET (col_a, col_b) = (col_a + 1, col_b + 1);
```

The following triggered action is legal because column **col_c** is not a column that the triggering event has updated:

```
UPDATE tab1 SET (col_c) = (col_c + 3);
```

In the preceding example, a triggered action on **col_a** or **col_b** would be illegal because a triggered action cannot be an UPDATE statement that references a column that was updated by the triggering event.

Important: Select triggers cannot be re-entrant triggers. If the triggering event is a SELECT statement, the triggered action cannot operate on the same table.

For a list of the rules that describe those situations in which a trigger can and cannot be re-entrant, see the CREATE TRIGGER statement in the *HCL® Informix® Guide to SQL: Syntax*.

INSTEAD OF triggers on views

A view is a synthetic table that you create with the CREATE VIEW statement and define with a SELECT statement. Each view consists of the set of rows and columns that the SELECT statement in the view definition returns each time you refer to the view in a query. To insert, update, or delete rows in the base tables of a view, you can define an INSTEAD OF trigger.

Unlike a trigger on a table, the INSTEAD OF trigger on a view causes HCL Informix® to ignore the triggering event, and instead perform only the triggered action.

For information on the CREATE VIEW statement and the INSTEAD OF trigger syntax and rules, including an example of an INSTEAD OF trigger that will insert rows on a view, see the *HCL® Informix® Guide to SQL: Syntax*.

INSTEAD OF trigger to update on a view

After you create one or more tables (like those named **dept** and **emp** in the following example), and then created a view (like the one named **manager_info**) from **dept** and **emp**, you can use an INSTEAD OF trigger to update that view.

The following CREATE TRIGGER statement creates **manager_info_update**, an INSTEAD OF trigger that is designed to update rows within the **dept** and **emp** tables through the **manager_info** view.

```
CREATE TRIGGER manager_info_update
   INSTEAD OF UPDATE ON manager_info
      REFERENCING NEW AS n
   FOR EACH ROW
      (EXECUTE PROCEDURE updtab (n.empno, n.empname, n.deptno,));

CREATE PROCEDURE updtab (eno INT, ename CHAR(20), dno INT,)
   DEFINE deptcode INT;
   UPDATE dept SET manager_num = eno where deptno = dno;
   SELECT deptno INTO deptcode FROM emp WHERE empno = eno;
   IF dno !=deptcode THEN
      UPDATE emp SET deptno = dno WHERE empno = eno;
   END IF;
   END PROCEDURE;
```

After the tables, view, trigger, and SPL routine have been created, the database server treats the following UPDATE statement as a triggering event:

```
UPDATE manager_info
   SET empno = 3666, empname = Steve
   WHERE deptno = 01;
```

This triggering UPDATE statement is not executed, but this event causes the trigger action to be executed instead, invoking the updtab() SPL routine. The UPDATE statements in the SPL routine update values into both the **emp** and **dept** base tables of the **manager_info** view.

Trace triggered actions

If a triggered action does not behave as you expect, place it in an SPL routine and use the SPL TRACE statement to monitor its operation. Before you start the trace, you must direct the output to a file with the SET DEBUG FILE TO statement.

Example of TRACE statements in an SPL routine

The following example shows TRACE statements that you add to the SPL routine **items_pct**. The SET DEBUG FILE TO statement directs the trace output to the file that the path name specifies. The TRACE ON statement begins tracing the statements and variables within the procedure.

```
CREATE PROCEDURE items_pct(mac CHAR(3))
DEFINE tp MONEY;
DEFINE mc_tot MONEY;
DEFINE pct DECIMAL;
SET DEBUG FILE TO 'pathname';

TRACE 'begin trace';
TRACE ON;
LET tp = (SELECT SUM(total_price) FROM items);
LET mc_tot = (SELECT SUM(total_price) FROM items
   WHERE manu_code = mac);
LET pct = mc_tot / tp;
IF pct > .10 THEN
   RAISE EXCEPTION -745;
END IF
TRACE OFF;
END PROCEDURE;

CREATE TRIGGER items_ins
INSERT ON items
REFERENCING NEW AS post_ins
FOR EACH ROW(EXECUTE PROCEDURE items_pct (post_ins.manu_code));
```

Example of TRACE output

The following example shows sample trace output from the **items_pct** procedure as it appears in the file that was named in the SET DEBUG FILE TO statement. The output reveals the values of procedure variables, procedure arguments, return values, and error codes.

```
trace expression :begin trace
trace on
expression:
  (select (sum total_price)
    from items)
evaluates to $18280.77 ;
let  tp = $18280.77
expression:
  (select (sum total_price)
    from items
    where (= manu_code, mac))
evaluates to $3008.00 ;
let  mc_tot = $3008.00
expression:(/ mc_tot, tp)
evaluates to 0.16
let  pct = 0.16
expression:(> pct, 0.1)
evaluates to 1
expression:(- 745)
evaluates to -745
raise exception :-745, 0, ''
exception : looking for handler
SQL error = -745 ISAM error = 0  error string =  = ''
exception : no appropriate handler
```

For more information about how to use the TRACE statement to diagnose logic errors in SPL routines, see Create and use SPL routines on page 1753.

Generate error messages

When a trigger fails because of an SQL statement, the database server returns the SQL error number that applies to the specific cause of the failure.

When the triggered action is an SPL routine, you can generate error messages for other error conditions with one of two reserved error numbers. The first one is error number -745, which has a generalized and fixed error message. The second one is error number -746, which allows you to supply the message text, up to a maximum of 70 bytes.

Summary

To introduce triggers, this chapter discussed the following topics:

- The components of the CREATE TRIGGER statement
- Types of DML statements that can be triggering events
- Types of SQL statements that can be triggered actions
- How to create BEFORE and AFTER triggered actions and how to use them to determine the impact of the triggering statement
- How to create a FOR EACH ROW triggered action and how to use the REFERENCING clause to refer to the values of columns both before and after the action of the triggering statement
- INSTEAD OF triggers on views, whose triggering event is ignored, but whose triggered actions can modify the base tables of the view
- The advantages of using SPL routines as triggered actions
- Special features of calls to trigger routines as triggered actions
- How to trace triggered actions if they behave unexpectedly
- How to generate two types of error messages within a triggered action.

Index

Special Characters

Brackets ([ ])
  range delimiters 977
([ ]), brackets
  array subscripts 711
  range delimiters 977
  substring operator 763, 1030, 1396, 1439
!=, not equal, relational operator 1549
?, question mark
  as placeholder in PREPARE 1718
.jar filename extension 234
' VERSION' table 1373
(_), underscore
  in SQL identifiers 1481
(;), semicolon
  list separator 1497, 1507
(:), colon
  cast (::) operator 1437, 1439
  list separator 1477, 1482, 1497, 1502, 1507
(!=), not equal to
  relational operator 1439
('), single quotation symbols
  string delimiter 1481
("), double quotation marks
  string delimiter 1404
("), double quotation symbols
  delimited SQL identifiers 1481
  string delimiter 1389, 1406, 1414
(()), parentheses
  delimiters in expressions 1428
({ }), braces
  collection delimiters 1195, 1404, 1406
  comment indicators 6, 1247
  pathname delimiters 1443
  specifying empty collection 1051
(@), at symbol 1227
(/), slash
  DATE separator 1399, 1428, 1469
  division operator 1424, 1439
  pathname delimiter 1445, 1474, 1502
(\ ), backslash
  invalid as delimiter 1472
  pathname delimiter 1447, 1499
(#), sharp
  comment indicator 1442
(%), percentage
  pathname indicator 1477
(<) less than symbol 1206
(<), less than
  angle (< >) brackets 1396
  relational operator 1439, 1471
(>), greater than
  angle (< >) brackets 1396
  relational operator 1322, 1439
(>) greater than symbol 1206
(|), pipe character 204, 592, 867
(|), vertical bar
  absolute value delimiter 1403
  concatenation (||) operator 1439
  field delimiter 1472
(||), concatenation operator 1003, 1021
($), dollar sign
  currency symbol 1406
  pathname indicator 1507
(_), underscore
  in SQL identifiers 1233
(--), double hyphen, comment indicator 6
(-), hyphen
  DATE separator 1469
  subtraction operator 1424, 1439
  symbol in syscolauth 1317, 1335
  symbol in sysfragauth 1347
  symbol in systabauth 1372
  unary operator 1425, 1439
(-), minus sign
  binary operator 1003
(,), comma
  list separator 1406, 1410, 1477
  thousands separator 1406
(::), cast operator 1022
(!), exclamation point 1206
  in smart-large-object filename 1098
(?), question mark
  as placeholder in PREPARE 619, 623
  as wildcard 977
  dynamic parameters 465
  generating unique large-object filename 1098
  variables in PUT 637
(.), decimal point
  INTERVAL separator 1194
  literal numbers 1198, 1205
(.), period
  DATE separator 1469, 1469
  decimal point 1400, 1406
  execution symbol 1442
  membership operator 1439
  nested dot notation 1431
(.), period symbol
  DECIMAL values 1199
  dot notation 1028
  MONEY values 1199
('), single quotation marks
  quoted string delimiter 1202
(‘), single quotation marks
  literal in a quoted string 1205
("), double quotation marks
  delimiting SQL identifiers 1203
  literal in a quoted string 1205
  quoted string delimiter 1202, 1205
  with delimited identifiers 1235, 1236
(), blank space
  padding VARCHAR values 1417
(*), asterisk
  all columns of a table 694, 875
  all fields of a collection variable 938
  all fields of a ROW column 708, 1028
  all fields of a ROW variable 719
  all labels of a security policy 675
  argument to COUNT 1159
  argument to the COUNT function 1158
  arithmetic operator 1003
  in Projection clause 694
  multiplication operator 1395, 1424, 1429, 1439
  systabauth value 1317, 1372
  wildcard character 977
  wildcard symbol 1333, 1385
(/), slash symbol
  arithmetic operator 1003
  UNIX path separator 1286
(/* */), slash and asterisk
  comment indicator 6, 626, 1247
(\), backslash
  as escape character 867
  as wildcard character 976
(\), backslash symbol
  Windows path separator 1286
(%), percent sign
  as wildcard 976
(^), caret
  as wildcard character 977
(+), plus sign
  addition operator 1424, 1439
  binary operator 1003
  in optimizer directives 1247
  truncation indicator 1488
  unary operator 1198, 1439
(=), equal sign
  relational operator 1206, 1208
(=), equality
  assignment operator 1447
  relational operator 1333, 1395, 1398, 1439
(~), tilde
  pathname indicator 1445
($), dollar sign
  in SQL identifiers 1233
<<label>> statement 913
=, equals, relational operator 1548, 1571
>=, greater than or equal to, relational operator 1549

A

A keyword
  in SELECT statement 711
AAO keyword
  in ALTER USER statement 165
  in CREATE DEFAULT USER statement 198
  in CREATE USER statement 416
Abbreviated year values 1467, 1469, 1469
ABS function 1057, 1058
ABSOLUTE keyword, in FETCH statement 513
ACCESS keyword 1423
  in ALTER TABLE statement 149
  in CREATE TABLE statement 352
  in GRANT statement 563
  in INFO statement 578
  in REVOKE statement 656, 675
Access method
  B-tree 1327, 1327, 1350, 1480
  built-in 1327, 1327
  default operator class, assigning 1267
  directives 1249
  index 237
  modifying 22
  primary 367, 1327, 1371, 1371
  privileges to alter 22
  privileges to create 184
  privileges to drop 470
  R-Tree 1480
  registering 184, 184
  secondary 237, 1327, 1341, 1352, 1408
  skip-scan 1249
  specifying for a table 367
  sysams data 1327
  sysindices data 1352
  sysopclasses data 1358
  systabamdata data 1371
Access mode for transactions 840
Access modes, description of 1749
Access privilege on UDRs 1810
ACCESS TIME keywords
  in ALTER TABLE statement 149

ACCESS TO keywords
in ALTER USER statement 165
in GRANT statement 568
ACCESS_METHOD keyword 1327
in ALTER ACCESS_METHOD statement 22
in CREATE ACCESS_METHOD statement 184
in DROP ACCESS_METHOD statement 469
ACCOUNT keyword
in CREATE USER statement 416
ACCOUNT LOCK keywords
in CREATE USER statement 416
ACCOUNT UNLOCK keywords
in CREATE USER statement 416
ACOS function 1108, 1110
ACOSH function 1111
Action clause, in CREATE TRIGGER statement
action list 399
syntax 392
Active connection 469, 780
ACTIVE keyword, in SAVE EXTERNAL DIRECTIVES statement 685
Active set
constructing with OPEN statement 617
definition of 1547, 1707
empty 521
of a cursor 1714
retrieving with FETCH 515
sequential cursor 441
ADD CONSTRAINT keywords
in ALTER TABLE statement 115
ADD CRCOLS keywords
in ALTER TABLE statement 98
ADD ERKEY keywords
in ALTER TABLE statement 98
ADD keyword
in ALTER ACCESS_METHOD statement 22
in ALTER FRAGMENT statement 47
in ALTER FUNCTION statement 82
in ALTER PROCEDURE statement 85
in ALTER ROUTINE statement 87
in ALTER SECURITY LABEL COMPONENT statement 88
in ALTER TABLE statement 100, 127
in ALTER TRUSTED CONTEXT statement 161
in ALTER USER statement 165
ADD REPLCHECK keywords
in ALTER TABLE statement 98
ADD ROWIDS keywords
in ALTER TABLE statement 98
ADD TYPE keywords
in ALTER TABLE statement 127
ADD USE FOR keywords
in ALTER TRUSTED CONTEXT statement 161
ADD VERCOLS keywords
in ALTER TABLE statement 98
ADD_MONTHS function 1101
Adding end-of-line character 220
Addition (+) operator 1424, 1439
ADDRESS keyword
in CREATE TRUSTED CONTEXT statement 413
ADDRESS keywords
in ALTER TRUSTED CONTEXT statement 161
ADMIN function 1146
AFTER keyword
in ALTER FRAGMENT statement 27, 47
in ALTER SECURITY LABEL COMPONENT statement 88
in CREATE TRIGGER statement 392, 399
Aggregate expressions
correlated 705, 706
Aggregate functions 1415
ALL, DISTINCT, and UNIQUE scope qualifiers 1159
and GROUP BY clause 1620
arguments 1155
arguments to COUNT 1159
as arguments 1211
AVG 1154, 1157, 1592
built-in 1404, 1406, 1414
COUNT 1154, 1158, 1593
description of 1592, 1600
finding NULL values 1710
in ESQL 1165, 1707
in EXISTS subquery 982
in expressions 705, 1592
in grid queries 1168
in GROUP BY Clause 753
in ORDER BY clause 762
in SELECT statement 705
in SPL routine 1775
in subquery 1640
MAX 1154, 1162, 1162, 1593
MIN 1154, 1162, 1593
no BYTE argument 1396
no collection arguments 1404, 1406, 1414
null value signalled 1705
RANGE 1163, 1594
standard deviation 1594
STDEV 1163, 1594
SUM 1154, 1163, 1595
summary 1165
syntax of built-in aggregates 1154
sysaggregates data 1326
user-defined 1326
using DISTINCT 1156
using UNIQUE 1156
VARIANCE 1164, 1595
AGGREGATE keyword
in CREATE AGGREGATE statement 185
in DROP AGGREGATE statement 470
AIX operating system 1486, 1503
Algebraic functions
ABS 1058
CEIL 1058
FLOOR 1058
MOD 1060
POW 1060
POWER 1060
ROOT 1060
ROUND 1061
SQRT 1061
TRUNC 1066
Alias
for a collection in SELECT statement 938
for a table in SELECT statement 713, 1212
for table name 1577
in optimizer directives 1247
to assign column names in temporary table 1625
using
as a query shortcut 1577
with a supertable 1591
with self-join 1625
Alias of a table 1317
Aliases for column names 753
Aliass
alias for a non-trivial column expression 768
Alignment of data type 1383
Alignment of data types 1332
ALL keyword 574, 728
aggregate scope qualifier 1159
beginning a subquery 736, 1639
in ALTER FRAGMENT statement 52, 57
in Condition segment 982
in CREATE VIEW statements 424
in DISCONNECT statement 467
in Expression segment 1154, 1166
in GRANT statement 552, 560, 563
in REVOKE FRAGMENT statement 681
in REVOKE statement 658, 666, 673, 675
in SELECT statement 698
in SET CONSTRAINTS statement 782
in SET Transaction Mode statement 841
in subquery 1639
with UNION operator 756, 758, 759
with UNION operator in SELECT 690
ALL operator 1439
ALL_ROWS keyword
in optimizer directives 1258
in SET OPTIMIZATION statement 820, 822
ALLOCATE COLLECTION statement 18
ALLOCATE DESCRIPTOR statement 19
ALLOCATE ROW statement 21
Allocating memory
for a collection variable 18
for system-descriptor area 19
ALLOW_NEWLINE configuration parameter 1204
allowed.surrogates file 165, 416
Allowing newline characters in quoted strings 1204
ALPHA class 1130, 1130
ALTER ACCESS_METHOD statement 22
ALTER FRAGMENT statement
am_readwrite purpose flag 1267
INIT clause 40
privileges required 23
restrictions 25
reverting to nonfragmented table
with DETACH 40
with INIT 42
syntax 23
with generalized-key index 40
ALTER FUNCTION statement 82
ALTER INDEX statement
reclustering a table 84
ALTER INDEX statement, locking table 1737
ALTER keyword
in GRANT statement 540, 551
in REVOKE statement 658, 665
Alter privilege 1317, 1372, 1386
ALTER privilege 540, 551, 658
ALTER PROCEDURE statement 85
ALTER ROUTINE statement 87
ALTER SECURITY LABEL COMPONENT statement 88
ALTER SEQUENCE statement 92, 1516
ALTER TABLE statement
casting effects 1434
changing column data type 139
changing data types 1389
dropping a column 133
lock mode 1484
next extent size 1322
privileges needed 95
restrictions

ADD clause 100
DROP Column clause 133
general 95
MODIFY clause 139
SERIAL columns 1412
SERIAL8 columns 1413
statistics options 157
synonyms 1516
ALTER TRUSTED CONTEXT statement 161
am_beginscan purpose task 1267
am_beginscan() function 1327
am_check purpose task 1267
am_close purpose task 1267
am_close() function 1327
am_cluster purpose flag 1267
am_create purpose task 1267
am_defopclass purpose value 1267
am_delete purpose task 1267
am_drop purpose task 1267
am_endscan purpose task 1267
am_expr_pushdown purpose flag 1267
am_getbyid purpose task 1267
am_getnext purpose task 1267
am_getnext() function 1327
am_insert purpose task 1267
am_insert() function 1327
am_keyscan purpose flag 1267
am_open purpose task 1267
am_open() function 1327
am_parallel purpose flag 1267
am_readwrite purpose flag 1267
am_rescan purpose task 1267
am_rowids purpose flag 1267
am_scancost purpose task 1267
am_sptype purpose value 1267
am_stats purpose task 1267
am_truncate access method 861
am_truncate purpose task 1267
am_unique purpose flag 1267
am_update purpose task 1267
Ambiguities in non-unique identifiers 1236
AND keyword
in BETWEEN condition 971
in Condition segment 967, 984
in OLAP window expressions 1189
with BETWEEN keyword 733
AND logical operator 1553
AND operator 1333, 1439
Angle bracket (<< ,,, >>) symbols
loop label delimiters 913
statement label delimiters 913
ANSI
isolation levels 1745
ANSI compliance
-ansi compilation flag 433
-ansi flag 292, 310, 370, 1466
comment symbols 6
creating views 421
DBANSIWARN environment variable 1466
DECIMAL range 1400
DECIMAL(p) data type 1399
Information Schema views 1384
isolation level 1388
list of SQL statements 14
privileges on UDRs 546
public synonyms 1371, 1373
renaming a table 648
table privileges 574
unbuffered logging 819
ansi flag 194
ANSI keyword
in CREATE DATABASE statement 190
ANSI-compliance
escape character 976
implicit transactions 833
isolation level 519, 840
owner names 1225
SQLSTATE codes 527
update cursors 437, 438, 456
warning after DELETE 458
ANSI-compliant database
creating 194
database object naming 1264
fixed-point DECIMAL values 194
FOR UPDATE not required in 1732
implicit transactions 451
opaque-type naming 264
procedure name 899
signalled in SQLWARN 1705
table access privileges 345
upshifting owner names 1265
warning after opening 430
with BEGIN WORK 169
ANSIOWNER environment variable 3, 194, 680, 1265, 1461
ANY keyword
in ALTER FRAGMENT statement 52, 57
in Condition segment 982
in SELECT statement 736
ANY keyword, in SELECT statement 1639
ANY operator 1439
APPEND keyword
in SET DEBUG FILE statement 797
Application
comment indicators 6
handling errors 1709
isolation level 1742
single-threaded 780
thread-safe 469, 469, 780, 782
update cursor 1746
Archiving
database server methods 1693
description of 1693
setting DBREMOTECMD 1477
transaction log 1693
Argument segment 1209
Arguments 1209
Arithmetic
DATE operands 1399, 1426
DATETIME operands 1425
integer operands 1395, 1403, 1403, 1415
INTERVAL operands 1426
operators 1439
time operands 1424
Arithmetic expressions 1562
Arithmetic operators
binary 1019
syntax 1003
unary 1019
Arithmetic operators, in expression 1562
ARRAY keyword
in ALTER SECURITY LABEL COMPONENT statement 88
in CREATE SECURITY LABEL COMPONENT statement 294
Array, with FETCH 517
AS keyword 694, 1212, 1437, 1437
Alias
for a table in DELETE statement 456
for a table in UPDATE statement 872
AS keyword
with column aliases 709
in ALTER FRAGMENT statement 27
in CONNECT statement 176
in CREATE CAST statement 188
in CREATE DISTINCT TYPE statement 199
in CREATE TABLE statement 315
in CREATE TRIGGER statement 380
Delete triggers 395
Insert triggers 396
Select triggers 398
Update triggers 397
in CREATE VIEW statement 421
in DELETE statement 451
in DELETE statements 456
in DROP CAST statement 471
in explicit casts 1022
in GRANT FRAGMENT statement 573
in GRANT statement 557
in Iterator segment 720
in MERGE statement 602
in Return Clause segment 1271
in REVOKE FRAGMENT statement 679
in REVOKE statement 654, 654, 670
in SELECT statement
in Iterator segment 720
in UPDATE statement 868
in UPDATE statements 872
with display labels 708, 709
AS PARTITION keywords, in ALTER FRAGMENT statement 27
AS REMAINDER keywords, in ALTER FRAGMENT statement 27
AS SELECT keywords
in CREATE TABLE statement 315
ASC keyword
in CREATE INDEX statement 241
in OLAP window expressions 1189
in SELECT statement 760, 764
order with nulls 764
Ascending order in SELECT 1540
Ascending sequence 94, 304
ASCII code points 1130
ASCII function 1120
ASIN function 1108, 1111
ASINH function 1111
Assign support function 264, 585, 593, 597, 877
assign() support function 1429
Associated statement 775
Asterisk (*)
argument to COUNT function 1158
arithmetic operator 1003
in C-style comment indicators 6
Projection clause 315, 694
wildcard character 977
Asterisk notation, in a SELECT statement 1585
Asterisk, wildcard character in SELECT 1538
AT keyword 1404
AT keyword, in INSERT statement 580
At symbol (@) 1225
ATAN function 1108, 1111
ATAN2 function 1108, 1112
ATANH function 1111
ATTACH keyword, in ALTER FRAGMENT statement 27
Attach list 31
Attached index 1480
Attached indexes 254, 778, 1348, 1469, 1512
ATTRIBUTES keyword
in ALTER TRUSTED CONTEXT statement 161

in CREATE TRUSTED CONTEXT statement 413
Audit Analysis officer 1500, 1500
AUDIT keyword
in CREATE TABLE statement 310, 346, 346
Audit-event mnemonics 1080
Auditing
selective row-level 346
AUTHENTICATION keyword
in ALTER TRUSTED CONTEXT statement 161
in CREATE TRUSTED CONTEXT statement 413
AUTHID keyword
in ALTER TRUSTED CONTEXT statement 161
in CREATE TRUSTED CONTEXT statement 413
Authorization identifier 282, 481, 831, 1042, 1238, 1262, 1379, 1388, 1664
AUTHORIZATION keyword
in ALTER USER statement 165
in CREATE DEFAULT USER statement 198
in CREATE SCHEMA statement 291
in CREATE USER statement 416
in GRANT statement 568
in SET SESSION AUTHORIZATION statement 831
AUTHORIZED keyword
in CREATE SECURITY POLICY statement 298
AUTO keyword
in ALTER TABLE statement 157
in CREATE TABLE statement 368
in UPDATE STATISTICS statement 883, 890
Auto Update Statistics (AUS) 888
AUTO_REPREPARE configuration parameter 26, 622, 634, 899
AUTO_STAT_MODE configuration parameter 368, 890, 1344
AUTO_STAT_MODE session environment setting 890, 1344
Autofree feature, in SET AUTOFREE 773
AUTOLOCATE configuration parameter 251, 357
AUTOLOCATE session environment option 251, 357
AVG function 1154, 1157
AVG function, as aggregate function 1592
AVG window function 1185
AVOID_EXECUTE keyword
in optimizer directives 1259
in SET EXPLAIN statement 800
AVOID_FACT keyword
in SET OPTIMIZATION statement 823
AVOID_FACT keyword, in optimizer directives 1256
AVOID_FULL keyword, in optimizer directives 1249
AVOID_HASH keyword, in optimizer directives 1255
AVOID_INDEX keyword, in optimizer directives 1249
AVOID_INDEX_SJ keyword, in optimizer directives 1249
AVOID_MULTI_INDEX keyword, in optimizer directives 1249
AVOID_NL keyword, in optimizer directives 1255
AVOID_STAR_JOIN keyword, in optimizer directives 1256
AVOID_STMT_CACHE keyword
in optimizer directives 1261

B

B abbreviation for byte 1146
B-tree access method 1327, 1350, 1480
B-tree cleaner list 889
B-tree index 1348
btree_ops operator class 270
default operator class 270
enforcing constraints 106
B-tree secondary-access method 267
B18030-2000 code set 777
Background mode 956
Backslash (\ ) symbol 1472
Backslash (\)
as escape character 204, 867, 977
Backup
file prefix 1490
Bandwidth 1491
BARGROUP keyword
in ALTER USER statement 165
in CREATE DEFAULT USER statement 198
in CREATE USER statement 416
BASE64 encoding of encrypted data 1081
BASED keyword
in CREATE TRUSTED CONTEXT statement 413
Batch file 958
BEFORE keyword
in ALTER FRAGMENT statement 27, 47
in ALTER SECURITY LABEL COMPONENT statement 88
in ALTER TABLE statement 100, 103
in CREATE TRIGGER statement 392, 399
BEGIN keyword 942
in Statement Block segment 1289
BEGIN WORK statement 168, 1693
BETWEEN keyword
in Condition segment 971
in OLAP window expressions 1189
using in WHERE clause 1547
using to specify a range of rows 1549
BETWEEN operator 1439
BigDecimal data type of Java 1281
BIGINT data type 1394
coltype code 1336
length (syscolumns) 1339
BIGSERIAL data type 1394
coltype code 1336
inserting values 585
invalid default 324
last BIGSERIAL value inserted 1613
last inserted value 1077
length (syscolumns) 1339
bin subdirectory 1443
Binary operator 186, 1019
Binding style 1387
Bit-hashing function 753
Bitshift functions 1069, 1070
Blank characters
in index names 476
in literal numbers 1198
in WHERE clause string literals 732
SPACE function 1125
BLOB data type 993
casting unavailable 1395
copying to a file 1098
copying to a smart large object 1099
creating from a file 1096
defined 1395
handle values 1031
inserting data 1395
size limit 994
storing 149, 352
syscolattribs data 1334
unloading 863, 865
unreferenced 860
BLOB keyword
in Data Type segment 993
Blobspace 39
Blobspaces
defined 1423
names 1481
sysblobs data 1332
BOOLEAN data type
defined 986, 1396
in Literal Row segment 1199
unloading 863
Boolean expression 967, 1553
with BOOLEAN data type 1396
with BYTE data type 1396
Boolean expression with TEXT data type 1415
Borland C compiler 1493
BOTH keyword, in TRIM expressions 1121
Bourne shell 1442, 1442
Braces ({})
collection delimiters 1195
comment indicator 6, 1247
specifying empty collection 1051
Braces ({}) comment delimiters 1761
Bracket ([]) symbols 1415
brackets substring 1415
BSON data type 700, 986, 987
BSON fields
indexing 243
BUCKETS keyword
in CREATE INDEX statement 249
BUFFERED keyword
in CREATE DATABASE statement 190
in SET LOG statement 819
BUFFERED LOG keyword
in CREATE DATABASE 194
Buffered logging 819
Buffers
BYTE or TEXT storage (DBBLOBBUF) 1466
fetch buffer (FET_BUFFER_SIZE) 1483
fetch buffer (SRV_FET_BUFFER_SIZE) 1512
floating-point display (DBFLTMASK) 1472
network buffer (IFX_NETBUF_SIZE) 1489
private network buffer pool 1488
Build table for hash joins 804
Built-in access method 1327, 1327
Built-in aggregates 1326, 1404, 1406, 1414
contrasted with user-defined 186
extending 186
Built-in casts 1332, 1434
Built-in data type, declaring variables 1766
Built-in data types
casts 1434, 1438
listed 1419
opaque 986
owner 199
privileges on 545
syntax 986
syscolumns.coltype code 1336
sysdistrib.type code 1344
sysxtdtypes data 1383
Built-in opaque data types 1437
Built-in roles
DBSECADM 558, 671
EXTEND 555

BY clause 1415
BY keyword 1396, 1415
in ALTER FRAGMENT statement 42, 45
in ALTER SEQUENCE statement 94
in CREATE INDEX statement 255
in CREATE SEQUENCE statement 304
in CREATE TABLE statement 356
in CREATE TEMP TABLE statement 375
in OLAP window expressions 1189
in SELECT statement 744, 752, 760
BY ORDER 1415
BYTE and TEXT columns, fragment storage 39
BYTE column
changing the data type 141
distribution statistics 889
BYTE data
effect of isolation on retrieval 816, 840
loading 593
storage location 993
unloading 863, 865
BYTE data type
casting to BLOB 1396
coltype code 1336
declaration syntax 993
defined 1396
increasing buffer size 1466
inserting values 1396
restrictions
in Boolean expression 1396
systables.npused 1373
with GROUP BY 1396
with LIKE or MATCHES 1396
with ORDER BY 1396
restrictions with GROUP BY 1620
selecting from BYTE columns 1396
setting buffer size 1466
sysblobs data 1332, 1332
syscolumns data 1339
sysfragments data 1348
using LENGTH function on 1610
with relational expression 1547
with SPL routines 921, 929
BYTE keyword
in Data Type segment 993
in Return Clause segment 1271

C

C compiler
default name 1493
INFORMIXC setting 1493
thread package 1515
C keyword
in GRANT statement 549
in REVOKE statement 664
C keyword, in External Routine Reference segment 1229
C shell 1442
.cshrc file 1442
.login file 1442
C++ API 1203
C++ map file 1497
Cache cohesion 835
CACHE keyword
in ALTER SEQUENCE statement 95
in CREATE SEQUENCE statement 305
in SET STATEMENT CACHE statement 833
Calculated expression
restrictions with GROUP BY 753
Calendar 1040
CALL keyword, in WHENEVER statement 904
CALL statement 915
CALL statement, in SPL function 1806
Callbacks 1278
CANNOTHASH modifier 753
Cardinality function
description of 1603
CARDINALITY function 1070, 1603
CARDINALITY() function 1404, 1406, 1414
Caret (^)
as wildcard character 977
use with brackets 977
Cartesian product 1254
basis of joins 1570
description of 1569
CASCADE keyword
in ALTER TABLE statement 109, 111
in CREATE TABLE statement 331
in DROP SECURITY POLICY statement 485
in DROP TABLE statement 489
in DROP VIEW statement 494
in REVOKE statement 654
Cascading deletes 1364
child tables 1683
definition of 1683
enabling in CREATE TABLE statement 111
locking associated with 454, 1683
logging 454, 1683, 1692
multiple child tables 454
referential integrity 1683
restriction 1684
Cascading Select trigger 1830
Cascading triggers
and triggering table 406
triggered actions 394
Case conversion
functions
INITCAP 1130
LOWER 1130
UPPER 1130
with INITCAP function 1606
with LOWER function 1605
with UPPER function 1605
CASE expression
description of 1565
in UPDATE statement 1677
using 1565
CASE expressions data type
compatibility 1033
CASE keyword
in CASE statement 916
in Expression segment 1033, 1034, 1035
Case sensitivity 195
CASE statement 916
Case-insensitive databases 1323, 1407, 1408
Cast (::) operator 1437, 1439
CAST AS keywords 1437
CAST keyword
in CREATE CAST statement 188
in DROP CAST statement 471
in explicit casts 1022
casting to CLOB 1415
Casts 1433, 1438
built-in 189, 471, 1332, 1434, 1437
creating 188
distinct data type 1438
dropping 471
explicit 189, 1022, 1332, 1437, 1437
from BYTE to BLOB 1396
function for 190
implicit 189, 1022, 1332, 1437, 1437
operator (::) 189, 1022
privileges 188
registering 188
rules of precedence 1437
symbol 1022
syscasts data 1332
user-defined (UDCs) 1332
Casts from TEXT 1415
cdr utility 728
cdrserver shadow column 98
cdrserver, replication column name 346
cdrsession
DBINFO function 1076
cdrtime shadow column 98
cdrtime, replication column name 346
CEIL function 1057, 1058
Chaining synonyms 309
CHAR data type
built-in casts 1436
converting to a DATE value 1600
converting to a DATETIME value 1602
defined 986
in INSERT 1205
in relational expressions 1547
substrings of 1546
truncation signalled 1705
CHAR_LENGTH function 1090, 1092
CHARACTER data type 1397
Character data types
Boolean comparisons 1417
casting between 1434
changing a column length or data type 142
data strings 1389
fixed and varying length 986
listed 1419
Character string
CHARACTER VARYING data type 1397
CLOB data type 1398
converting to a DATE value 1600
converting to a DATETIME value 1602
DATETIME literals 1427
LVARCHAR data type 1405
NCHAR data type 1407
NVARCHAR data type 1408
VARCHAR data type 1417
with DELIMIDENT set 1481
CHARACTER VARYING data type
defined 1397
CHARACTER_LENGTH function 1090, 1092
Character-based applications 1500, 1514
CHARINDEX function 1134
Check constraints
creation-time value 1469, 1471
defining 335
reject files 208
syschecks data 1333
syscheckudrdep data 1333
syscoldepend data 1335
sysconstraints data 1341
Check constraints, definition of 1681
CHECK keyword
in ALTER TABLE statement 112
in CREATE TABLE statement 335
in CREATE VIEW statement 421, 425
Child table 111, 120, 602
chkenv utility 1442
error message 1444
syntax 1444
CHR function 1130
Chunks 1423
Class libraries, shared 1529
CLASS_ORIGIN keyword, in GET DIAGNOSTICS statement 531

CLIEN_LABEL environment variable 1463
Client APIs 1203
CLIENT_LOCALE environment variable 777, 1469
CLIENT_TZ keyword
  in SET ENVIRONMENT statement 1044
Client/server
  DataBlade API 1423
  default database 1498
  INFORMIXSQLHOSTS environment variable 1499
  shared memory communication segments 1498
  stacksize for client session 1499
CLIENTBINVAL data type 700, 986
CLOB data type
  casting unavailable 1398
  code-set conversion 1398
  collation 1398
  copying to a file 1098
  copying to a smart large object 1099
  creating from a file 1096
  defined 1398
  handle values 1031
  inserting data 1398
  multibyte characters 1398
  size limit 994
  storing 149, 352
  syscolattribs data 1334
  unloading 863, 865
  unreferenced 860
CLOB keyword
  in Data Type segment 993
CLOB TEXT 1415
CLOSE DATABASE statement 173
CLOSE DATABASE statement, effect on database locks 1736
CLOSE statement 1506
  closing a collection cursor 173
  closing a select cursor 172
  closing an insert cursor 172
  cursors affected by transaction end 173
  syntax 170
CLUSTER keyword
  in ALTER INDEX statement 84
  in CREATE INDEX statement 239
Clustered index 84, 239
Clustering 1327, 1348, 1352
Clustering, specifying support for 1267
CMCONFIG environment variable 1462
COALESCE function 1036
Code set 777
Code sets
  conversion 1449, 1449
  EBCDIC 1388
  ISO 8859-1 1346
Collation 1415
  CLOB data type 1398
  GL_COLLATE table 1373
  localized 775
  NCHAR data type 1407
  NVARCHAR data type 1408
  server_attribute data 1388
COLLATION keyword, in SET COLLATION statement 775
Collection constructors
  example 1051, 1052
  restrictions 1051
Collection cursor
  closing 173
  DECLARE for ESQL/C variable 445
  declaring 938
  defined 445
  in SPL 938
  inserting into 520, 639, 639
  INTO clause 520
  opening 619
Collection data type 1001, 1001
  allocating memory 18
  casting matrix 1438
  defined 1431
  defining a column 1001
  deleting 457
  element, searching for with IN 973
  empty 1431
  IN operator 973
  LIST 1001, 1404
  loading 596
  MULTISET 1001, 1406
  returning number of elements 1070
  selecting from 718
  SET 1001, 1414
  sysattrtypes data 1330
  sysxtddesc data 1382
  sysxtdtypes data 1382, 1383
  unloading 863
  updating 1219
COLLECTION data type
  coltype code 1336
Collection data types
  accessing 1581, 1586
  counting elements in 1603, 1603
  description of 1586
  element, searching for with IN 1587
  simple 1586
  updating 1675, 1675
  using the CARDINALITY function 1603
collection delimiters 1414, 1431
COLLECTION keyword
  in ALLOCATE COLLECTION statement 18
  in DEALLOCATE COLLECTION statement 431
  untyped collection variable 718
Collection subquery
  description of 1645
  ITEM keyword 1646, 1647
  using ITEM keyword in 1647
Collection Subquery segment 965
Collection types
  in an SPL routine 1761
  in DELETE statement 1661
Collection values, inserting into columns 1668
Collection variable
  accessing 1219
  accessing values 1219
  associating cursor with 445
  cursor for 520
  deallocating memory for 431
  defining, restrictions on 1767
  in SELECT statement 718
  manipulating values 1219
  nested 1586, 1587
  opening a cursor 619
  selecting 1587
  selecting from 718
  selecting, inserting elements 445
  untyped 18, 431
  updating 520, 1219
  with DESCRIBE INPUT statement 466
  with DESCRIBE statement 462
collection-derived table
  using in SPL 1796
Collection-derived table 1212, 1648
  accessing elements in a collection 1648
  collection cursor 448, 520, 639
  collection variables with 1219
  description of 1645, 1792
  FOREACH statement 939
  in SELECT statement 1219
  INSERT statement with 448, 592
  restrictions on 1648
  ROW data types
    selecting fields 719
  row types in 1215
  row variables with 1223
  SELECT statement with 718
    fields from row variable 719
  TABLE keyword 1212, 1219, 1223
  UPDATE statement with 882, 1219, 1223
  where allowed 1223
Collection-Derived Table segment 1212
Collections
  accessing a nested collection 1223
  accessing elements 1213
  allocating memory 18
  constructors 1051
  deleting elements from 457
  example of deleting elements 1220
  example of inserting elements 1222
  example of updating 1222
  generating values for 1051
  inserting values into 586
  nested 1197
  restrictions when accessing elements 1214
  restrictions when defining 1003
  restrictions with inserting null values 586
  selecting from 718
  specifying literal values 1196
  untyped 18
Collections, with INSERT statement 1668
Colon
  cast (::) operator 1437
Colon symbol (:)
  cast operator (::) 1022
  with database qualifier 1226
Color and intensity screen attributes 1500
Column alias
  in Projection clause 709
  in SELECT statement 709
Column definition clause 320, 322
COLUMN keyword
  in ALTER TABLE statement 113, 144
  in CREATE TABLE statement 320, 322
COLUMN keyword, in RENAME COLUMN statement 643
Column name
  alias 753
  alias in FROM clause of SELECT 711
  dot notation 1028
  using functions as names 1236, 1237
  using keywords as names 1237
Column number, using 1542
Column substring 1030
Column-level encryption 1080
Column-level privileges
  systabauth data 1317
  systabauth table 1372
Columns
  adding 100
  adding a NOT NULL constraint 143
  adding a PRIMARY KEY constraint 143
  changing data type 1389, 1433
  changing the data type 143

check constraints 335
constraints (sysconstraints) 1341
declaring aliases 315
default values (sysdefaults) 1342
defining as primary key 330
definition of 1534
descending order 1540
description of 1528
distribution statistics 889
dropping 133, 134
expression 704, 1025
for a column expression in SELECT statements 768
hashed 1348
in relational model 1528
in superstores_demo database 1517
in trigger-event definitions 1828
inserting BLOB data 1395
inserting into 581
label on 1653
modifying with ALTER TABLE 139
number, effect on triggers 387
order in Projection list 694
ordering the selection of 1539
primary or foreign key 330
privileges 540
projection 1028
range of values 1341
referenced and referencing 331
removing a NOT NULL constraint 143
renaming 643
row-type, definition of 1583
shadow columns 98
specifying a subscript 763, 1030
syscolumns data 1336
virtual 424
columns Information Schema view 1384
COLUMNS keyword, in INFO statement 578
Combine function 1326
COMBINE keyword
in CREATE AGGREGATE statement 185
Comma (,) symbol in pathnames 1097
Command file 6
Comment indicator 1442
Comment lines 1442
Comment symbol
braces ({ }) 6
double hyphen (--) 6
in application programs 6
in optimizer directives 1247
in prepared statements 626
slash and asterisk (/* */) 6
COMMIT WORK statement
closing cursors 1751
in ANSI-compliant databases 176
in non-ANSI databases 175
releasing locks 1741, 1751
setting SQLCODE 1724
syntax 175
COMMITTED keyword
in SET ISOLATION statement 812, 814
in SET TRANSACTION statement 839
Committed read 1388
Committed Read isolation level 814
Committed Read isolation level (Informix) 1744
COMMITTED READ keywords
in SET ISOLATION statement 812
Communications support module 1494
commutator function
definition 1760
Commutator function 1360
Companion functions 503, 1277
Compare support function 264
Comparison condition, description of 1547
Compiling
ESQL/C programs 1461
INFORMIXC setting 1493
JAVA_COMPILER setting 1502
multithreaded ESQL/C applications 1515
Complete-connection level settings
of SET EXPLAIN 806
of SET ISOLATION 813
of SET LOCK MODE 819
Complex data type 1429, 1432
collection types 1431
invalid in distributed queries 700
loading element values 596
ROW types 1432
sysattrtypes data 1330
unloading 866
Complex table expression 714
Complex view 408
Compliance
ANSI/ISO standard for SQL 1384, 1466
sql_languages.conformance 1387
X/Open CAE standards 1384
XPG4 standard 1386
COMPONENT keyword
in ALTER SECURITY LABEL COMPONENT statement 88
in CREATE SECURITY LABEL COMPONENT statement 294
in CREATE SECURITY LABEL statement 293
in DROP SECURITY statement 485
in RENAME SECURITY statement 647
COMPONENTS keyword
in CREATE SECURITY POLICY statement 298
Composite index 1350
Composite key 341
Compound assignment 943
Compound query 1650
Compressed indexes
creating 252
COMPRESSED keyword
in CREATE INDEX statement 252
in CREATE TABLE statement 365
COMPRESSED option
in CREATE TABLE statement 350
Compressed tables
creating 365
CONCAT function 1114
CONCAT() operator function 1021
Concatenation (||) operator 1439
Concatenation operator (||) 1003, 1021
concsm.cfg file 1494
Concurrency
access modes 1749
active set 1715
ANSI isolation levels 1743
Cursor Stability isolation (Informix) 1744
database lock 1736
deadlock 1750
description of 1694, 1733
Informix isolation levels 1743
isolation level 1742
kinds of locks 1735
lock duration 1741
lock scope 1736
multiple programs 1734
table lock 1737
with CREATE INDEX 260
with DROP INDEX 477
with SET ISOLATION 812
with SET TRANSACTION 840
with START VIOLATIONS TABLE 846
CONCURRENT keyword, in CONNECT statement 176
Condition segment
BETWEEN keyword 971
DELETING 975
INSERTING 975
IS NOT NULL 974
IS NULL 974
join conditions 736
keywords
ALL 982
ANY 982
EXISTS 982
LIKE 975
MATCHES 975
NOT 976
SOME 982
null values 984
SELECTING 975
subquery in SELECT 979
syntax 967
UPDATING 975
Conditional expressions
CASE 1032
COALESCE 1032
DECODE 1032
NVL 1032
Conditions
comparison 969
IN operator 972
NOT IN operator 972
Configuration file
.cshrc file 1442
.informix 1442, 1444, 1482, 1484
.login file 1442
.profile file 1442
for communications support module 1494
for connectivity 1498, 1499
for database servers 1482, 1503
for terminal I/O 1500
Configuration parameters
ALLOW_NEWLINE 1204
AUTO_REPREPARE 26, 622, 634, 899
AUTO_STAT_MODE 368, 890
DATASKIP 796
DB_LIBRARY_PATH 1285
DBCREATE_PERMISSION 190
DBSERVERALIASES 176, 457, 587, 702, 877
DBSERVERNAME 457, 587, 702, 877, 1043
DBSPACETEMP 375, 1477
DEADLOCK_TIMEOUT 477
DEF_TABLE_LOCKMODE 137, 366, 602, 1484
DEFAULTESCCHAR 204
DIRECTIVES 1246, 1485
DISABLE_B162428_XA_FIX 1492
DS_NONPDQ_QUERY_MEM 762
EXPLAIN_STAT 804
EXT_DIRECTIVES 686, 1261, 1343, 1485

IFX_EXTEND_ROLE 225, 236, 549, 555, 664, 668
ISOLATION_LOCKS 1744
LOCKS 472, 601
MAX_PDQPRIORITY 827
MITRACE_OFF 1376, 1377
OPT_GOAL 1258, 1506
OPTCOMPIND 834, 1255, 1504
PN_STAGEBLOB_THRESHOLD 357
RESIDENT 1486
SBSPACENAME 149, 352, 911, 1086, 1087
SEQ_CACHE_SIZE 95, 305
shared memory base 1492
SQL_LOGICAL_CHAR 102, 142, 928, 1373
STACKSIZE 1280, 1499
STMT_CACHE 834, 1513
STMT_CACHE_HITS 836
STMT_CACHE_NOLIMIT 836
STMT_CACHE_SIZE 836
STOP_APPLY 817
SYSSBSPACENAME 157, 368, 890
TEMPTAB_NOLOG 374
UPDATABLE_SECONDARY 817
USELASTCOMMITTED 600, 839
USEOSTIME 1044
USERMAPPING 165, 198, 568
USTLOW_SAMPLE 892
CONNECT BY clause
Hierarchical clause 738
in SELECT statement 744
CONNECT DEFAULT statement 1498
CONNECT keyword 539
in REVOKE statement 656
in SELECT statement 744
Connect privilege 1322, 1379
granting 539
revoking 656
CONNECT statement 176, 535, 1474, 1496, 1498
CONNECT_BY_ISLEAF keyword
in SELECT statement 746
CONNECT_BY_ROOT operator 746
CONNECTION keyword
in CREATE TRUSTED CONTEXT statement 413
CONNECTION keyword, in SET CONNECTION statement 779
CONNECTION_NAME field 535
CONNECTION_NAME keyword, in GET DIAGNOSTICS statement 531
Connections
active 469, 780
closing 173
context 467, 779
current 176, 468, 782
default 181, 468, 781
dormant 176, 467, 779
explicit 182, 472
IFX_VALIDATE_STORAGEPOOLPATH environment variable 1492
implicit 173, 182, 190, 429, 468, 781
INFORMIXCONRETRY environment variable 1495
INFORMIXCONTIME environment variable 1496
INFORMIXSERVER environment variable 1498
returning connection names 534
single-threaded applications 780
specifying connection names 180, 781
Connectivity information 1499
considerations for MERGE 612
Constant expression 1040
in SELECT 704
inserting with PUT 636
Constant expressions 1664
CONSTRAINT keyword
in ALTER TABLE statement 108, 115, 117, 130
in CREATE TABLE statement 336
in RENAME CONSTRAINT statement 644
Constraints
adding foreign key 120
adding primary-key 127
adding referential 127
adding to a column with data 148
adding unique 127
adding with ALTER TABLE 115, 117
affected by dropping a column from table 134
check
creation-time value 1471
syschecks data 1333
syscheckudrdep data 1333
syscoldepend data 1335
checking 411, 602, 841
column
sysconstraints data 1341
detached checking 342
disabled 115, 337, 793
dropping 130
enabled 115, 337
encountering violations while adding 127
filtering 337
filtering to violations table 793
foreign key 123, 331, 670
limit on size 339
mode 123, 337
modifying columns that have constraints 139
multiple-column 339
name 336
not null
collection data types 1406, 1414, 1431
NOT NULL 328
collection data types 1404
syscoldepend data 1335
syscolumns data 1336
sysconstraints data 1341
NOT NULLy 330
number of columns allowed 339
object mode 1357
primary key 330
sysconstraints data 1341
sysreferences data 1364
unique SERIAL values 1412
unique SERIAL8 values 1412
privileges needed to create 127
referential 120, 123, 331, 853
sysconstraints data 1341
sysreferences data 1364
renaming 644, 645
restrictions 108
single-column 327
suspending validation 123
system catalog tables 336
table
sysconstraints data 1341
transaction mode 841
unique
sysconstraints data 1341
sysviolations data 1380
validating 119, 790
violations 1380
CONSTRAINTS keyword
in SET CONSTRAINTS statement 782
in SET Database Object Mode statement 786, 787
in SET Transaction Mode statement 841
Constraints, entity integrity 1681
constrid column 40
Constructors 1414, 1431
functions
collections 1051
row 1050
CONTEXT keyword
in CREATE TRUSTED CONTEXT statement 413
in RENAME TRUSTED CONTEXT statement 650
CONTEXT keywords
in ALTER TRUSTED CONTEXT statement 161
in DROP TRUSTED CONTEXT statement 492
CONTINUE keyword
in WHENEVER statement 904
CONTINUE statement 919
Conversion function, description of 1600
Converting data types
DATE and DATETIME 1436
INTEGER and DATE 1436
number and string 1436
number to number 1435
retyping a column 1433
Coordinated deletes 1725
Coordinated Universal Time (UTC) 1078, 1078, 1613
Correlated aggregate expression 705, 706
Correlated column reference 705, 706
Correlated subquery 714
defined 979
definition of 1635
restriction with cascading deletes 1684
Correlation name 713
declaring 230, 276, 395
in routines 407
qualifying values 403
scope of reference 402
COS function 1108, 1109
COSH function 1109
COSTFUNC keyword, in Routine Modifier segment 1275
COUNT DISTINCT function 1158
COUNT field
in system-descriptor area 20
with DESCRIBE INPUT statement 465
with DESCRIBE statement 461
COUNT function
and GROUP BY 1620
as aggregate function 1593
count rows to delete 1660
defined 1158
restriction with CREATE TRIGGER statement 402
syntax 1154
use in a subquery 1662
with DISTINCT 1593
COUNT UNIQUE function 1158
COUNT window function 1185
CPFIRST environment variable 1461
CPU cost 1512
CPU usage cost 1278

CPU VP class 1276
CRCOLS keyword
in ALTER TABLE statement 98
in CREATE TABLE statement 310, 346, 346
CREATE ACCESS_METHOD statement 184, 1327
CREATE AGGREGATE statement 185
CREATE CAST statement 188, 1332, 1436
CREATE DATABASE statement 1474
ANSI compliance 194
NLSCASE case sensitivity 195
setting shared lock 1736
SQLWARN after 1705
syntax 190
using with PREPARE 190
CREATE DEFAULT USER statement 198
CREATE DISTINCT TYPE statement 199, 1383, 1401, 1518
CREATE EXTERNAL TABLE statement 201
adding a newline 221
adding an end-of-line character 220
delimited format example 213, 216
fixed format example 216
refreshing a data warehouse table 218
CREATE FUNCTION FROM statement 236
CREATE FUNCTION FROM statement, in embedded languages 1763
CREATE FUNCTION statement 225, 1365
inside CREATE FUNCTION FROM statement 1763
using 1754
WITH LISTING IN clause 1814
CREATE FUNCTION, return clause 1758
CREATE IMPLICIT CAST statement 1518
CREATE INDEX statement 1350, 1352, 1373, 1480
cluster with fragments 239
composite indexes 245
compressed 252
disabled indexes 259
index-type options 239
sort order 246
specifying object modes 258
storage options 251, 1480
CREATE INDEX statement, locking table 1737
CREATE OPAQUE TYPE statement 263, 1408
CREATE OPCLASS statement 267
CREATE OPERATOR CLASS statement 1358
CREATE PROCEDURE FROM statement 281
CREATE PROCEDURE FROM statement, in embedded languages 1763
CREATE PROCEDURE statement 270, 1365, 1503
inside CREATE PROCEDURE FROM 1763
using 1754
WITH LISTING IN clause 1814
CREATE ROLE statement 282, 1365
CREATE ROUTINE FROM statement 284, 1365, 1503
CREATE ROW TYPE statement 286, 1336, 1409
CREATE SCHEMA statement 291, 1317
CREATE SECURITY LABEL COMPONENT statement 294
CREATE SECURITY LABEL statement 293
CREATE SECURITY POLICY statement 298
CREATE SEQUENCE statement 302, 1369
CREATE SYNONYM statement 305, 1370, 1371
CREATE TABLE statement
access-method option 367
assigning data types 1389
cascading deletes 1683
collection types 1586
column definition clause 320
constraints
check 335
composite keys 339, 341
defining 327
distinct 329
example 341
NULL 329
restrictions 328
unique 329
creating composite columns 339
default lock mode 1484
default privileges 1503
defining constraints 339
fragmenting
by expression 358
round-robin 357
hierarchy 1589
locking options 365
multiset columns 1646
newline example 221
ON DELETE CASCADE clause 1659
primary keys 1682
PUT clause 352
row type columns 1581
SET constructor 1414
setting the lock mode 1740
smart large object columns 1603
specifying cascading deletes 334
specifying column-default values 324
specifying storage location 350
statistics options 368
syntax 310
typed table 1581
typed tables 1409
WITH ROWIDS keywords 356
CREATE TEMP TABLE statement 370, 1477
column-level constraints 372
CREATE TRIGGER statement 378, 1378, 1824
CREATE TRUSTED CONTEXT statement 413
CREATE USER statement 416
CREATE VIEW statement 421, 714, 1317, 1380
CREATE XADATASOURCE statement 426, 1381
CREATE XADATASOURCE TYPE statement 428
CREATE XADATASOURCETYPE statement 1381
Creation-time settings of environment variables 399
Cross join 1571
CROSS keyword in SELECT statement 721
Cross-database DML operations 412, 457, 587, 612, 702, 877
Cross-database SQL operations 1697
Cross-server connection requirements 1700
Cross-server DML operations 412, 877
Cross-server SQL operations 1697
CRPT audit-event mnemonic 1080
CSN encryption 1079
CTRL-J
newline
preserving in quoted strings 1204
Currency symbol 1406
Current database, specifying with DATABASE 429
Current date 1044, 1342, 1467
CURRENT DORMANT keywords, in SET CONNECTION statement 779
CURRENT function
as an argument 1044
as constant expression 1040, 1664
comparing column values 1596
in ALTER TABLE statement 103
in Condition segment 972
in CREATE TABLE statement 324
in DEFINE statement 922
in INSERT statement 588, 588
in WHERE condition 1044
using 1596
CURRENT keyword 1424
in DELETE statement 451
in DISCONNECT statement 468
in FETCH statement 513
in OLAP window expressions 1189
in SET CONNECTION statement 779
in UPDATE statement 881
CURRENT ROW keywords
in OLAP window expressions 1189
CURRENT_ROLE operator
defined 1042
syntax 1040
usage 1042
CURRENT_USER function
in ALTER TABLE statement 103
in DEFINE statement 922
in INSERT statement 588
CURRENT_USER operator
syntax 1040
CURRVAL operator 302, 1046, 1048
Cursor
activating with OPEN 616
active set of 1714
affected by transaction end 173
characteristics 441
closing 170, 1751
closing with ROLLBACK WORK 683
declaring 433, 1711
definition of 1711
direct 172
end of transaction 1751
for insert 1727
for update 1731, 1741
restricted statements 444
using in ANSI-mode databases 444, 445
using in non-ANSI databases 444, 445
freeing automatically with SET AUTOFREE 773
implicit 935
manipulation statements 12
opening 617, 1712, 1714
prepared statement with 445
read-only
restricted statements 444
using in ANSI-mode databases 444, 445
using in non-ANSI databases 444, 445
retrieving values with FETCH 513, 1712
scroll 1713
sequence of program operations 436, 1711
sequential 935, 1713, 1715
sqlca.sqlcode value 1071
stability 837
statement identifier with 445
types of 616
WITH HOLD examples 442
with INTO keyword in SELECT 710
with transactions 449
Cursor function 935, 935, 1274

CURSOR keyword
in DECLARE statement 433, 449
in SET ISOLATION statement 812
Cursor Stability isolation level 814
Cursor Stability isolation level (
Informix
)
1744
CURSOR STABILITY keywords
in SET ISOLATION statement 812
CYCLE keyword
in ALTER SEQUENCE statement 95
in CREATE SEQUENCE statement 304
Cyclic query 1684

D

Dangling child records 454
Data
access statements 12
automatic compression 365
data definition statements 9
data manipulation statements 11
encryption 1079
inserting with LOAD 592
integrity statements 11
Data buffering 522
data column of sysprocbody table 1081
Data corruption 1322, 1334
Data definition statements 1720
Data dependencies
syscheckudrdep data 1333
syscoldepend data 1335
sysdepend data 1343
Data dictionary 1316
Data distributions 1322, 1344, 1479
confidence level 894
on temporary tables 885
refreshing only stale statistics 157, 368, 890
STATCHANGE threshold for selectively updating 157, 368, 890
Data encryption functions 1617
DATA field
with DESCRIBE INPUT statement 465
with DESCRIBE statement 461
Data integrity 1387, 1680
failures 1691
Data models, description of 1521
Data pages 1334, 1350, 1373
Data replication 1694
data type collation 1415
data type restrictions 1415
data type restrictions in Boolean expression 1415
Data type segment 985
data type UPDATE statements 1415
Data types 612, 985
alignment 200
approximate 1386
automatic conversions 1708
BIGINT 1394
BIGSERIAL 1394
BLOB 1395
BOOLEAN 1396
BYTE 1396
casting 188, 1022, 1433, 1438
changing with ALTER TABLE 143
CHARACTER 1397
CHARACTER VARYING 1397
classified by category 1389
CLOB 1398
collection 1001, 1431
collection, accessing 1581, 1586
complex 998, 1429
considerations for INSERT 584, 1205
conversion 1433, 1664, 1708
conversion costs 1206
DATE 1399
DEC 1399
DECIMAL 1399
distinct 996, 1432
DISTINCT 1401
DOUBLE PRECISION 1402
exact numeric 1386
extended 1429
fixed point 1400
FLOAT 1402
floating-point 1399, 1402, 1415
IDSSECURITYLABEL 992, 1403, 1420
inheritance 1409
INT 1403
INT8 1403
INTEGER 1403
internal 1389
Label-based access control (LBAC) 992
length (syscolumns) 1339
LIST 1404
LVARCHAR 1405
MONEY 1406
MULTISET 1406
named ROW 1409
NCHAR 1407, 1407
NUMERIC 1408
NVARCHAR 1408
opaque 263, 1433
OPAQUE 1408
Opaque data types
smart large objects 1423
promotion 1081
REAL 1409
representation 200
returned by UDFs 1272
ROW 1409, 1410
sequential integer 1412
SERIAL 1412
SERIAL8 1412
SET 1414
simple large object 993, 1273, 1422
SMALLFLOAT 1415
SMALLINT 1415
smart large object 994, 1423
specifying with CREATE VIEW 421
summary list 1389
unique numeric value 1412
unnamed ROW 1410
VARCHAR 1417
Data warehouse tables
initial load 217
loading from other database servers 219
refreshing tables
periodically 218
Data-integrity violations 844, 856
Data-type promotion 1115, 1132, 1420
Database
transaction log 192
unlogged 192
Database administrator 539
Database administrator (DBA)
granting privileges 537
revoking privileges 656
Database identifiers 1481
DATABASE keyword
in CLOSE DATABASE statement 173
in CREATE DATABASE statement 190
in DATABASE statement 429
in DROP DATABASE statement 472
in RENAME DATABASE statement 645
Database object
constraints as a 1684
index as a 1684
naming 1225
object modes 1684
owner 1262
trigger as a 1684
violation detection 1684
Database object mode
examples 1685
for triggers 383, 791
privileges required 785
specifying 784
Database Object Name segment 1225, 1225
Database Server Administrator 1229
Database server administrator (DBSA) 1317
Database Server Administrator (DBSA) 225, 282, 475, 480, 483, 555, 556, 668, 1500
Database servers
archiving 1693
attributes in Information Schema view 1388
code set 1388
default connection 1498
default isolation level 1388
identifying host computer name 1613
identifying version number 1613
locking tables 1737
optimizing queries 1506
pathname for 1474
remote 1483
returning the SQL identifier 1043
role separation 1500
server name 1342, 1474
signalled in SQLWARN 1705
statement caching 1752
DATABASE statement 1474
determining database type 430
exclusive mode 431
locking 1736
specifying current database 429
SQLWARN after 430, 1705
syntax 429
Database statements 182
database-level 539
Database-level privilege 556
not available for roles 656
revoking 656
Databases
ANSI-compliant 430, 1531
closing with CLOSE DATABASE 173
compared to file 1522
concurrent use 1524
control of 1524
data types 1389
data warehousing 319
Databases
superstores_demo 1517
default isolation levels 815, 840
definition of 1528
demonstration databases
superstores_demo 1517
dropping 472
external 1226, 1696
global variables 921
identifiers 1481
isolation level 837

joins in stores_demo 1516
lock 431
locking 1736
management of 1527
modifying contents of 1524
naming conventions 1224
nonlogging 840
nonlogging database 410
object-relational 1517
object-relational, description of 1529
objects, sysobjstate data 1357
OLTP 319
opening in exclusive mode 431
optimizing queries 885
privileges 1379
relational, description of 1527
remote 1225, 1696
renaming 645
running in secondary mode 430
server 1524
stores_demo 1516
superstores_demo
demonstration database 1517
syscrd 1317
sysmaster 1317
sysutils 1317
sysuuid 1317
DataBlade API 1278
DataBlade API (LIBDMI) 1203
DataBlade modules 270, 1529
Client and Server API 1423
data types (sysbuiltintypes) 1317
trace messages (systracemsgs) 1376, 1377
user messages (syserrors) 1346
DATAFILES keyword
in CREATE EXTERNAL TABLE statement 202
DATASKIP configuration parameter 796
DATASKIP keyword
in SET DATASKIP statement 796
DATE data type
abbreviated year values 1467
casting to integer 1436
coltype code 1336
converting to a character string 1601
converting to DATETIME 1436
declaration syntax 995
defined 1399
display format 1469
functions 1101
functions returning 1596
in expressions 1424, 1426
in ORDER BY sequence 1540
in relational expressions 1547
international date formats 1399
source data 1426
DATE function 1101, 1101, 1102
DATE function, as conversion function 1600
DATE() function 1427, 1469
DATETIME data type
abbreviated year values 1467
as quoted string 1205
cast to Coordinated Universal Time (UTC) 1078
coltype code 1336
converting to a character string 1601
converting to DATE 1436, 1436
declaration syntax 995
displaying format 1599
EXTEND function 1426
extending precision 1425
functions 1101
functions returning 1596
in expressions 1424, 1428
in INSERT 1205
in ORDER BY sequence 1540
in relational expressions 1547
length (syscolumns) 1339
literal values 1045
source data 1427
DATETIME values, formatting 1599
DAY function 1101, 1103, 1596
DAY keyword 1194, 1194, 1194, 1194
UNITS operator 1399, 1428
Day of the week 1104
DB_LIBRARY_PATH configuration parameter 1285
DB_LOCALE environment variable 195, 430, 775, 777, 804, 1233
DB-Access
creating database with 1720
DB-Access utility 1322, 1385, 1445, 1472, 1474, 1478, 1498
DBA keyword 539
in CREATE FUNCTION statement 225
in CREATE PROCEDURE statement 270
in REVOKE statement 656
DBA privilege 1346, 1376, 1377, 1379
with CREATE ACCESS_METHOD statement 184
with CREATE SCHEMA 291
with DROP DATABASE 472
with DROP TRIGGER statement 491
with REVOKE statement 656
DBA routines 1360
DBA-privileged UDR 228, 229, 274, 275
DBA. 539
DBACCNOIGN environment variable 169, 1465, 1466
DBANSIWARN environment variable 194, 292, 310, 370, 421, 1466
DBBLOBBUF environment variable 595, 865, 1466, 1466
DBCENTURY environment variable 35, 399, 593
defined 1467
effect on functionality of DBDATE 1469
expanding abbreviated years 1468
DBCREATE_PERMISSION configuration parameter 190
DBDATE environment variable 863, 1061, 1205, 1399, 1469, 1664
DBDATE keyword
in CREATE EXTERNAL TABLE statement 204
DBDELIMITER environment variable 597, 867, 1472
DBEDIT environment variable 1472
dbexport utility 817, 1472
DBFLTMASK environment variable 1472
dbhostname option of DBINFO 1077
DBINFO function 1071, 1076
DBINFO function, in SELECT statement 1613
dbinfo utility 1473
DBINFO_DBSPACE_RETURN_NULL_FOR_INVALID_PARTNUM environment variable 1473
DBLANG environment variable 1473, 1473
dbload utility 1395, 1396, 1415, 1472
DBMONEY environment variable 593, 863, 1199, 1406
DBMONEY keyword
in CREATE EXTERNAL TABLE statement 204
DBPATH environment variable 179, 182, 1474
DBPRINT environment variable 1476
DBREMOTECMD environment variable 1477
DBSA group 556
DBSA keyword
in ALTER USER statement 165
in CREATE DEFAULT USER statement 198
in CREATE USER statement 416
DBSA. 555
dbschema utility 94, 1360
DBSECADM keyword
in GRANT statement 558
in REVOKE statement 671
DBSECADM role 40, 100, 113, 413, 831, 1403, 1420
dbsendrecv data type 264
DBSERVERALIASES configuration parameter 176, 457, 587, 608, 702, 877, 1700
DBSERVERNAME configuration parameter 457, 587, 608, 702, 877, 1700
DBSERVERNAME function
constant expression 1043
in ALTER TABLE statement 103
in Condition segment 972
in CREATE TABLE statement 324
in DEFINE statement 922
DBSERVERNAME function, in INSERT statement 1664
DBSERVERNAME function, in SELECT statement 1612
dbservername.cmd batch file 1447
dbspace
for BYTE or TEXT values 1332
for system catalog 1317
for table fragments 1347
for temporary tables 1477
name 1481
renaming with onspaces 23
the database dbspace 1273
dbspace, name returned by DBINFO function 1613
dbspaces
number 1090
skipping if unavailable 796
DBSPACETEMP configuration parameter 375, 1477
DBSPACETEMP environment variable 257, 375, 898, 929, 1477
DBSSO keyword
in ALTER USER statement 165
in CREATE DEFAULT USER statement 198
in CREATE USER statement 416
DBTEMP environment variable 1478
DBTIME environment variable 593, 863
DBUPSPACE environment variable 895, 898, 1479
DDL (Data Definition Language) statements
listed 9
Deadlock 519
Deadlock detection 819, 1750
DEADLOCK_TIMEOUT configuration parameter 477
DEADLOCK_TIMEOUT setting in ONCONFIG 819
DEALLOCATE COLLECTION statement 431, 431
DEALLOCATE DESCRIPTOR statement 432
DEALLOCATE ROW statement 433
DEBUG keyword

in SET DEBUG FILE statement 797
Debugging SPL routines 911
DEC data type 1399
DECIMAL data type
built-in casts 1435, 1436
coltype code 1336
defined 1399
disk storage 1400
display format 1472
fixed point 1400
floating point 1399
length (syscolumns) 1339
literal values 1199
DECIMAL data type, signalled in SQLWARN 1705
Decimal digits, display of 1472
Decimal point
DBFLTMASK setting 1472
DECIMAL radix 1400
Decimal point (.)
INTERVAL separator 1194
literal numbers 1198, 1205
declaration syntax 993, 993, 993
Declarative statements 950
DECLARE CURSOR statement 1727
DECLARE statement 1506
cursor characteristics 441
CURSOR keyword 441
cursors with prepared statements 445
cursors with transactions 449
description of 1711
FOR INSERT clause 1727
FOR UPDATE 1731
FOR UPDATE keywords 438
function cursor 436
insert cursor 440, 444
restrictions with SELECT with ORDER BY 766
SCROLL keyword 442, 1713
select cursor 436
syntax in ESQL/C routines 433
syntax in SPL routines 449
updating specified columns 438
WHERE CURRENT OF clause 438
WITH HOLD clause 1751
with SELECT statement 711
DECODE function 1038, 1614
DECRYPT_BINARY function 1086, 1398, 1617
DECRYPT_CHAR function 1086, 1398, 1617
DEF_TABLE_LOCKMODE configuration parameter 136, 137, 451, 602, 1484
DEF_TABLES_LOCKMODE configuration parameter 1740
Default code set 1130
Default connection 181, 468
Default database locale 1323
Default database server 181, 468, 534
Default escape character 867
Default isolation level 840
DEFAULT keyword
in ALTER TABLE statement 103
in ALTER TRUSTED CONTEXT statement 161
in CONNECT statement 176
in CREATE DEFAULT USER statement 198
in CREATE EXTERNAL TABLE statement 204
in CREATE TABLE statement 324
in CREATE TRUSTED CONTEXT statement 413
in CREATE USER statement 416
in DISCONNECT statement 468
in GRANT statement 554
in REVOKE statement 654, 668
in SET CONNECTION statement 779, 781
in SET OPTIMIZATION statement 823
in SET PDQPRIORITY statement 827
in SET ROLE statement 830, 830
Default role 282, 554, 668
DEFAULT ROLE keywords
in ALTER TRUSTED CONTEXT statement 161, 161
in CREATE TRUSTED CONTEXT statement 413, 413
Default user ID 181
Default values
in column 1681
using 1710
DEFAULT_ATTACH environment variable 254, 1480
DEFAULT_ROLE operator
defined 1043
syntax 1040
usage 1043
DEFAULTESCCHAR configuration parameter 204
Defaults
C compiler 1493
century 1467
character set for SQL identifiers 1481
compilation order 1461
configuration file 1503
connection 1498
data type 1411
database server 1474, 1498
DATE display format 1399
DATE separator 1469
DECIMAL precision 1399
disk space for sorting 1479
fetch buffer size 1483
heap size 1502
index storage location 1480
isolation level 1388
join method 1504
level of parallelism 1508
lock mode 1484
message directory 1473
MONEY scale 1406
operator class 1327, 1358
printing program 1476
query optimizer goal 1506
sysdefaults.default 1342
table privileges 1503
temporary dbspace 1477
terminfo direcotry 1514
text editor 1472
Deferred extent allocation 351
DEFERRED keyword
in SET CONSTRAINTS statement 782
WITH ERROR keywords
in SET CONSTRAINTS statement 782
WITHOUT ERROR keywords
in SET CONSTRAINTS statement 782
DEFERRED keyword, in SET Transaction Mode statement 841
DEFERRED_PREPARE keyword
in SET DEFERRED_PREPARE statement 799
Deferred-Prepare feature 799
DEFINE keyword, in Statement Block segment 1289
DEFINE statement
default value clause 922
syntax 920
DEFINE statement of SPL 1412, 1412
defined Data types 1415
Degrees
converting degrees to radians 1113
converting radians to degrees 1112
DEGREES function 1108
DELETE CASCADE keywords
in ALTER TABLE statement 111
Delete clause in MERGE statement 602
DELETE keyword 574
in CREATE TABLE statement 331
in CREATE TRIGGER statement 386
in GRANT statement 540
in MERGE statement 602
in REVOKE FRAGMENT statement 681
in REVOKE statement 658
Delete MERGE operations 1662
Delete privilege 540, 658, 1347, 1372, 1503
DELETE statement 1380
DELETE statements 1322
and triggers 400
cascading 454
collection columns with 457
collection types 1661
coordinated deletes 1725
count of rows 1723
cursor with 438
description of 1659
developing 1662
distributed 457
duplicate rows 1729
embedded 1702, 1723
lock mode 1747
number of rows 1704
OUT parameter 1153
preparing 1718
remove all rows 1659
row types 1661
selected rows 1660
specific rows 1660
syntax 451
transactions with 1724
using 1723
using subquery 1662
WHERE clause restriction 1662
with cursor 1725
with SELECT . . . FOR UPDATE 771
with supertables 1661
with update cursor 456
within a transaction 451
Delete trigger 385, 1378
Delete using TRUNCATE 1659
Deleting from a specific table in a table hierarchy 453
DELETING operator 230, 975
DELIMIDENT environment variable 177, 451, 861, 911, 1203, 1205, 1233, 1233, 1235, 1481
Delimited format
loading 216
unloading 219
Delimited identifiers 1481, 1481
in database server names 1227
multibyte characters 1235
non-ASCII characters 1235
DELIMITED keyword
in CREATE EXTERNAL TABLE statement 204
Delimiter
for fields 1472
for identifiers 1481

for LOAD input file 597
specifying with UNLOAD 867
DELIMITER keyword
in CREATE EXTERNAL TABLE statement 204
in LOAD statement 592
in UNLOAD statement 863
DELUXE keyword
in CREATE EXTERNAL TABLE statement 204
DELUXE mode load 215
demonstration databases
stores_demo 1516
Demonstration databases
tables 1517
DENSE_RANK function 1178
DENSERANK function 1178
DES3 (Triple Data Encryption Standard) 1087
DESC keyword
in CREATE INDEX statement 241
in OLAP window expressions 1189
in SELECT statement 760, 764
order with nulls 764
Descending index 1350
Descending order in SELECT 1540
Descending sequence 94, 304
DESCRIBE INPUT statement 463
DESCRIBE statement 1491
collection variable with 462
syntax 459
Describe-for-updates 1491
DESCRIPTOR keyword
in ALLOCATE DESCRIPTOR statement 19, 19
in DEALLOCATE DESCRIPTOR statement 432
in DESCRIBE INPUT statement 465
in DESCRIBE statement 459, 461
in EXECUTE statement 498, 501
in FETCH statement 513
in OPEN statement 616
in PUT statement 634
Descriptors of simple large objects 1273
destroy() support function 264, 457, 490, 861, 1429
DETACH keyword
in ALTER FRAGMENT statement 52, 57
DETACH keyword, in ALTER FRAGMENT statement 36
Detached index 47, 255, 1480
Detached statement 775
Diagnostics area 526
DIAGNOSTICS keyword, in GET DIAGNOSTICS statement 526
Diagnostics table 1380
creating 844
declaring a name 846
default name 846
description of 1688
example of privileges 1690
examples 857
examples of starting 1688
filtering mode 793
how to stop 856
relationship to target table 848
relationship to violations table 848
restriction on dropping 491
schema 853
Difference set operation 1656
Direct cursor 172
DIRECTIVES configuration parameter 1246, 1485
Directives for query optimization 1485, 1504, 1506
DIRECTIVES keyword, in SAVE EXTERNAL DIRECTIVES statement 685
Dirty Read isolation level 260, 476, 477, 813
Dirty Read isolation level (
Informix
)
1743
DIRTY READ keywords
in SET ISOLATION statement 812
DISABLE keyword
in ALTER TRUSTED CONTEXT statement 161
in CREATE TRUSTED CONTEXT statement 413
Disabled database objects 1380
DISABLED keyword
in ALTER FRAGMENT statement 52
in ALTER TABLE statement 108, 117, 120
in CREATE INDEX statement 258
in CREATE TABLE statement 336, 337
in CREATE TRIGGER statement 383, 383
in SET AUTOFREE statement 773
in SET CONSTRAINTS statement 782
in SET Database Object Mode statement 784, 788, 791
in SET DEFERRED_PREPARE statement 799
in SET INDEXES statement 810
DISCARD keyword
in ALTER FRAGMENT statement 52, 57
DISCONNECT ALL statement 535
DISCONNECT statement 467, 535
DISK keyword
in CREATE EXTERNAL TABLE statement 202
Disk space
for data distributions 1479
for temporary data 1477
Display label
in ORDER BY clause 1567
with SELECT 1564
Display labels
in CREATE VIEW statement 423
in Projection clause 708
in SELECT statement 708
Distinct data types 996
base types 996
casting 200
casts 1438
casts and DROP TYPE 493
creating with CREATE DISTINCT TYPE 199
distributed queries 700, 702
dropping 493
privileges 199, 545, 662
restrictions on source type 199
sysxtdtypes data 1383
Usage privilege 545
DISTINCT data types
defined 1401
distributed queries 996
sysxtddesc data 1382
sysxtdtypes data 1383, 1401
DISTINCT keyword
aggregate scope qualifier 1159
in ALTER TABLE statement 106, 117
in CREATE DISTINCT TYPE statement 199
in CREATE INDEX statement 239
in CREATE TABLE statement 327, 339
in CREATE TEMP TABLE statement 372, 373
in Expression segment 1154, 1166
in OLAP window aggregate expressions 1185
in SELECT statement 698
in subquery 981
relation to GROUP BY 1620
using in SELECT 1542
using with COUNT function 1593
Distinct-type variable 1769
Distributed Computing Environment (DCE) 1515
Distributed DML operations 457, 587, 877
Distributed queries 700, 702, 1429, 1483
cross-database queries 700
cross-server queries 702
Distributed SQL operations 1697
Distribution bins 895
Distributions 895
dropping 893, 893
privileges required to create 894
DISTRIBUTIONS keyword, in UPDATE STATISTICS statement 883, 895
divide() operator function 1019
Division (/) symbol, arithmetic operator 1003
DML (Data Manipulation Language) statements 11
DOCUMENT clause, use in SPL routine 1761
DOCUMENT keyword
in CREATE FUNCTION statement 225
in CREATE PROCEDURE statement 270
Dollar ($) symbol
in SQL identifiers 1233
White space characters
delimited identifiers 1233
Dollar ($) sign 1406
Dollar ($) symbol
in MONEY values 1205
in prepared statements 626
in SQL identifiers 1231
prefix for C variables 1286
prefix in pathnames 1286
Domain name 413
Domain of column 1681
DORMANT keyword, in SET CONNECTION statement 779
Dot notation 1028, 1584
double (C) data type 1402
double hyphen (--) comment indicator 1761
Double hyphen (--) comment indicator 6, 1247
Double quotation marks (")
delimiting SQL identifiers 1203
literal in a quoted string 1205
quoted string delimiter 1202, 1205
Double-precision floating-point number 1402
DRDA client-server communication protocol 527, 612
DROP ACCESS_METHOD statement 469
DROP AGGREGATE statement 470
DROP ALL ROLLING keywords in ALTER FRAGMENT statement 52, 57
DROP CAST statement 471, 1518
DROP CONSTRAINT keywords, in ALTER TABLE statement 130
DROP CRCOLS keywords
in ALTER TABLE statement 98
DROP DATABASE statement 472, 1474
DROP DISTRIBUTIONS keywords, in UPDATE STATISTICS statement 883
DROP ERKEY keywords

in ALTER TABLE statement 98
DROP FUNCTION statement 473, 1360
DROP INDEX statement 476, 1373
DROP INDEX statement, locking table 1737
DROP keyword
in ALTER ACCESS_METHOD statement 22
in ALTER FRAGMENT statement 50
in ALTER FUNCTION statement 82
in ALTER PROCEDURE statement 85
in ALTER ROUTINE statement 87
in ALTER TABLE statement 133, 144
in ALTER TRUSTED CONTEXT statement 161
in ALTER USER statement 165
in TRUNCATE statement 858
in UPDATE STATISTICS statement 883
DROP OPCLASS statement 478
DROP PARTITION keywords, in ALTER FRAGMENT statement 50
DROP PROCEDURE statement 479, 1360
DROP REPLCHECK keywords
in ALTER TABLE statement 98
DROP ROLE statement 481
DROP ROUTINE statement 481, 1360
DROP ROW TYPE statement 483, 1409
DROP ROWIDS keywords
in ALTER TABLE statement 98
DROP SECURITY LABEL COMPONENT statement 485
DROP SECURITY LABEL statement 485
DROP SECURITY POLICY statement 485
DROP SECURITY statement 485
DROP SEQUENCE statement 487, 1516
DROP SPECIFIC FUNCTION statement 473
DROP SPECIFIC PROCEDURE statement 479
DROP STORAGE keywords, in TRUNCATE statement 860
DROP SYNONYM statement 488
DROP TABLE statement 489, 1516
DROP TRIGGER statement 491
DROP TRUSTED CONTEXT statement 492
DROP TYPE statement 493, 1401, 1408
DROP USE FOR keywords
in ALTER TRUSTED CONTEXT statement 161
DROP USER statement 494
DROP VERCOLS keywords
in ALTER TABLE statement 98
DROP VIEW statement 494, 1385, 1516
DROP XADATASOURCE statement 495
DROP XADATASOURCE TYPE statement 496
DS_MAX_QUERIES configuration parameter 827
DS_NONPDQ_QUERY_MEM configuration parameter 762
Duplicate values
allowing or excluding 698, 756, 758, 759
in a UNION ALL query 756, 758, 759
in case-insensitive databases 699
Duplicate values, finding 1568
Dynamic cursor names 433
Dynamic link library 1229
Dynamic log feature 956
Dynamic management statement 12
Dynamic parameters 465
Dynamic routine-name specification 506
for SPL function 1808
for SPL routine 1808
of SPL functions 506
of SPL procedures 513
rules for 1809
Dynamic SQL
description of 1702, 1717
freeing prepared statements 1720

E

EACH keyword, in CREATE TRIGGER statement 392, 399
East Asian locales 1092
EBCDIC collation 1388
Editor, DBEDIT setting 1472
ELIF keyword, in IF statement 941
ELSE keyword
in CASE statement 916
in Expression segment 1034, 1035
in IF statement 941
EMACS text editor 1472
Embedded SQL
definition of 1701
languages available 1701
Empty set 1431
Empty strings 732
ENABLE keyword
in ALTER TRUSTED CONTEXT statement 161
ENABLE keyword, in CREATE TRUSTED CONTEXT statement 413
ENABLED keyword
CASCADE keyword
in ALTER TABLE statement 108
DELETE keyword
in ALTER TABLE statement 108
in ALTER FRAGMENT statement 52
in ALTER TABLE statement 108
in CREATE INDEX statement 258
in CREATE TABLE statement 336, 337
in CREATE TRIGGER statement 383
in SET AUTOFREE statement 773
in SET CONSTRAINTS statement 782, 790
in SET Database Object Mode statement 788, 790, 791
in SET DEFERRED_PREPARE statement 799
in SET INDEXES statement 810
ON keyword
in ALTER TABLE statement 108
ENABLED NOVALIDATE keywords
in SET Database Object Mode statement 784
ENABLED NOVALIDATE mode 792
ENCRYPT_AES function 1617
ENCRYPT_DES function 1398
ENCRYPT_TDES function 1087, 1398, 1617
Encrypted data 1079, 1081
Encryption and decryption functions
DECRYPT_BINARY 1086
DECRYPT_CHAR 1086
ENCRYPT_TDES 1087
GETHINT 1088
syntax 1079
Encryption communication support module (ENCCSM) 1079
END CASE keywords, in CASE statement 916
END EXCEPTION keywords, in ON EXCEPTION statement 949
END FOR keywords, in FOR statement 931
END FOREACH keywords, in FOREACH statement 935
END FUNCTION keyword
in CREATE FUNCTION statement 225
END IF keywords, in IF statement 941
END keyword
in Expression segment 1034, 1035
in Statement Block segment 1289
End of data
signal in SQLCODE 1704, 1709
signal only for SELECT 1730
SQLCODE 1712
when opening cursor 1712
END PROCEDURE keywords
in CREATE PROCEDURE statement 270
END WHILE keywords, in WHILE statement 945, 961
Enterprise Replication 1317
creating shadow columns 346, 347, 347
Entity integrity 1681
env utility 1444
ENVIGNORE environment variable
defined 1442, 1482
relation to chkenv utility 1444
Environment configuration file
debugging with chkenv 1444
setting environment variables in UNIX 1441, 1442
ENVIRONMENT keyword
in SET OPTIMIZATION statement 820, 823
Environment variables 1044
ANSIOWNER 3, 680, 1265, 1461
CLIENT_LABEL 1463
CLIENT_LOCALE 777, 1469, 1469
CMCONFIG 1462
Colon
pathname separator 1502
command-line utilities 1446
CPFIRST 1461
DB_LOCALE 195, 296, 430, 775, 777, 804, 1233
DBACCNOIGN 169, 1465, 1466
DBANSIWARN 194, 292, 421, 433, 1466
DBBLOBBUF 595, 865, 1466, 1466
DBCENTURY 35, 399, 593, 1467
DBDATE 593, 863, 1061, 1205, 1399, 1469
DBDELIMITER 597, 867, 1472
DBEDIT 1472
DBFLTMASK 1472
DBINFO_DBSPACE_RETURN_NULL_FOR_INVALID_PARTNUM 1
DBLANG 1473
DBMONEY 593, 863, 1406
DBPATH 182, 1474
DBPRINT 1476
DBREMOTECMD 1477
DBSPACETEMP 257, 375, 898, 1477, 1477
DBTEMP 1478
DBTIME 593, 863
DBUPSPACE 895, 898, 1479
DEFAULT_ATTACH 254, 1480
DELIMIDENT 177, 451, 861, 1203, 1233, 1235, 1244, 1481
displaying current settings 1444, 1447
ENVIGNORE 1482
FET_BUF_SIZE 1483
GL_DATE 593, 863, 1205, 1399, 1469
GL_DATETIME 584, 592, 593, 863, 1469
GL_USEGLU 777
how to set
in Bourne shell 1443
in C shell 1443
in Korn shell 1443
how to set in Bourne shell 1443
how to set in Korn shell 1443
IFMXMONGOAUTH 1483
IFX_DEF_TABLE_LOCKMODE 137, 137, 366, 1484
IFX_DIRECTIVES 1246, 1485

IFX_DIRTY_WAIT 600, 859
IFX_EXTDIRECTIVES 686, 1261, 1343, 1485
IFX_LARGE_PAGES 1486
IFX_LOB_XFERSIZE 1487
IFX_LONGID 1231, 1488
IFX_NETBUF_PVTPOOL_SIZE 1488
IFX_NETBUF_SIZE 1489
IFX_NO_SECURITY_CHECK 1489
IFX_NO_TIMELIMIT_WARNING 1490
IFX_NODBPROC 1490
IFX_NOT_STRICT_THOUS_SEP 1490
IFX_ONTAPE_FILE_PREFIX 1490
IFX_PAD_VARCHAR 1491
IFX_TABLE_LOCKMODE 602
IFX_UPDDESC 460, 463, 1491
IFX_VALIDATE_STORAGEPOOLPATH 1492
IFX_XASTDCOMPLIANCE_XAEND 1492
IFX_XFER_SHMBASE 1492
INF_ROLE_SEP 1500, 1500
INFORMIXC 1493
INFORMIXCMCONUNITNAME 1494
INFORMIXCMNAME 1493
INFORMIXCONCSMCFG 1494
INFORMIXCONRETRY 1495
INFORMIXCONTIME 1496
INFORMIXCPPMAP 1497
INFORMIXDIR 1497
INFORMIXSERVER 181, 182, 1043, 1498
INFORMIXSHMBASE 1498
INFORMIXSQLHOSTS 1499, 1499
INFORMIXSTACKSIZE 1499
INFORMIXTERM 1500
INTERACTIVE_DESKTOP_OFF 1501
JAR_TEMP_PATH 1501
JAVA_COMPILER 1502
JVM_MAX_HEAP_SIZE 1502
LD_LIBRARY_PATH 1502
LIBPATH 1503
limitations 1441
listed by topic 1449
manipulating in Windows environments 1445
modifying settings 1443
NODEFDAC 228, 229, 274, 275, 345, 662, 1503
ONCONFIG 1503
ONINIT_STDOUT 1504
OPT_GOAL 1258, 1506
OPTCOMPIND 834, 1504
OPTMSG 1505
OPTOFC 1506
overriding a setting 1442, 1482
PATH 1507, 1507
Pathname
  for client or shared libraries 1502
PDQPRIORITY 827, 827, 834, 1507
PSM_ACT_LOG 1508, 1508
PSM_DBS_POOL 1509
PSM_DEBUG 1509
PSM_DEBUG_LOG 1510
PSM_LOG_POOL 1510
PSORT_DBTEMP 1510
PSORT_NPROCS 1511
RTREE_COST_ADJUST_VALUE 1512
rules of precedence in UNIX 1445
rules of precedence in Windows 1448
scope of reference 1446
setting 1446
  at the command line 1441
  in a configuration file 1441
  in a login file 1441
  in a shell file 1442
  in Windows environments 1445
  with the System applet 1446
setting in autoexec.bat 1447
setting with SYSTEM statement 958
SRV_FET_BUF_SIZE 1512
standard UNIX system 1440
STMT_CACHE 834, 1513
TABLE_SIZE 377
TERM 1514
TERMCAP 1514
TERMINFO 1515
THREADLIB 1515
types of 1440
unsetting 1443, 1447, 1481
USE_DTENV 584, 592
USETABLENAME 95, 489, 489, 489, 859, 1516
view current setting 1444
where to set 1442
Environment variablesDB_LOCALE
DB_LOCALE 195, 296, 430, 775, 777, 804, 1233
Equal sign (=)
relational operator 1206, 1208
equal() operator function 329, 1208
equal() support function 1429
Equality (=) operator 1398
Equals (=) relational operator 1548, 1571
Equi-join 1571, 1571
ERKEY keyword
in ALTER TABLE statement 98
in CREATE TABLE statement 310, 346, 347
Error checking
continuing after error in SPL routine 952
Deadlock 956
error status with ON EXCEPTION 949
Long transaction rollback 956
simulating errors 1819
SPL routines 1817, 1820
with SYSTEM 956
with WHENEVER 906
ERROR keyword
in ALTER TABLE statement 108
in CREATE INDEX statement 258
in CREATE TABLE statement 336, 337
in SET CONSTRAINTS statement 782
in SET Database Object Mode statement 784, 788
in SET INDEXES statement 810
in WHENEVER statement 904
synonym for SQLERROR 906
Error message files 1473, 1706
Error messages
for trigger failure 1834
generating in a trigger 1834
Errors
after DELETE 1724
codes for 1704
dealing with 1709
detected on opening cursor 1712
during updates 1691
inserting with a cursor 1728
ISAM error code 1704
ESCAPE keyword
in Condition segment 975
in CREATE EXTERNAL TABLE statement 204
with LIKE keyword 734, 979
with MATCHES keyword 734, 979
ESCAPE keyword, using in WHERE clause 1558
esql command 1461, 1493
esql compiler 1247
ESQL/C
collection cursor with FETCH 520
collection cursor with PUT 639
cursor example 442
cursor use 1711, 1716
deallocating collection-variable memory 431
deallocating row-variable memory 433
DELETE statement in 1723
delimiting host variables 1702
dynamic embedding 1702, 1717
error checking for aggregate functions 1165
error handling 1709
esqlc command 1461
fetching rows from cursor 1712
host variable 1702, 1703
indicator variable 1709
INSERT in 1727
inserting collection variables with 586
inserting row variables 586, 588
long identifiers 1488
message chaining 1505
multithreaded applications 1515
overview 1700, 1723, 1723
preprocessor 1701
program compilation order 1461
scroll cursor 1713
selecting single rows 1707
SQL Communications Area 1703
SQL statements valid only in ESQL/C 1021
SQLCODE 1703
SQLERRD fields 1704
statements valid only in ESQL/C 1021
static embedding 1701
UPDATE in 1731
ESQL/C API 1203
Exact numeric data types 1386
EXCEPT operator
in SELECT statement 690, 756, 759, 760
restrictions on use 756
Exception handler 1292
EXCEPTION keyword
in GET DIAGNOSTICS statement 531
in ON EXCEPTION statement 949
in RAISE EXCEPTION statement 953
Exclamation point (!) 1206
in smart-large-object filename 1098
EXCLUSIVE keyword
in DATABASE statement 429, 1736
in LOCK TABLE statement 598
Exclusive lock 1735
Exclusive lock mode 237, 260, 477, 598
Executable file location 1285
Executable programs 1507
Executable statements 950
EXECUTE FUNCTION keywords 935
in DECLARE statement 433
in INSERT statement 590
in Statement Block segment 1289
EXECUTE FUNCTION statement 503
and triggers 400
how it works 503
preparing 505
syntax 502
with SPL 1804
EXECUTE IMMEDIATE statement 506

restricted statement types 507
EXECUTE IMMEDIATE statement, description of 1720
EXECUTE ON keywords
in GRANT statement 546, 548, 549
in REVOKE statement 662
Execute privilege 548, 549, 1358, 1503
DBA keyword, effect of 1813
objects referenced by a routine 1812
EXECUTE PROCEDURE keywords 935
in DECLARE statement 433
in INSERT statement 590
in Statement Block segment 1289
EXECUTE PROCEDURE statement 935
in triggered action 400
syntax 510
with SPL 1804
EXECUTE statement
INTO clause 498
INTO SQL DESCRIPTOR clause 500
syntax 496
USING DESCRIPTOR clause 502
EXEMPTION keyword
in GRANT statement 560
in REVOKE statement 673
EXISTS keyword 1655
beginning a subquery 735
in a WHERE clause 1639
in Condition segment 982
in Condition subquery 982
in SELECT statement 1642
EXIT statement 929, 929
EXP function 1088
EXPLAIN keyword
in optimizer directives 1259
SET EXPLAIN statement 800
explain out location 804
explain output 803
Explain output 806
explain output file 803, 804, 1479
EXPLAIN_STAT configuration parameter 804
Explicit cast 1332, 1437
EXPLICIT keyword
in CREATE CAST statement 188
Explicit pathnames 1447, 1476
Explicit temporary tables 1477
Exponent 1400
Exponential function 1088
Exponential notation 1399
Exponential number 1199
Export support function 264
export utility 1443
export_binary() support function 1429
export( ) support function 863
export() support function 1429
Exportbinary support function 264
exportbinary( ) support function 863
EXPRESS keyword
in CREATE EXTERNAL TABLE statement 204
Express-mode load
procedure 213
Expression
Boolean 969
CASE 1565
casting 1022
constant 1040
date-oriented 1596
description of 1562
display label for 1564
in SPL routine 1775
list of 1005
ordering by 765
smart large objects in 1031
Expression Fragment Clause 358
EXPRESSION keyword
in ALTER FRAGMENT statement 42, 42, 45
in CREATE INDEX statement 255
in CREATE TABLE statement 356
Expression segment
aggregate expressions 1154
cast expressions 1022
column expressions 1025
combined expressions 1019
list of expressions 1005
syntax 1003
Expression-based fragmentation 1348, 1469, 1471
EXT_DIRECTIVES configuration parameter 686, 1261, 1343, 1485
EXTDIRECTIVES keyword, in SET ENVIRONMENT statement 686, 1261
EXTEND function 1101, 1106, 1426
using in expression 1599
with DATE and DATETIME values 1596
EXTEND keyword
in GRANT statement 555
in REVOKE statement 668
EXTEND role 225, 236, 279, 281, 282, 284, 475, 480, 481, 483, 1810
Extended data types 1383, 1429, 1429, 1518
Extensibility, description of 1529
Extensible Markup Language (XML) 1398
Extension checking (DBANSIWARN) 1466
EXTENT keyword
in ALTER TABLE statement 149
in CREATE TEMP TABLE statement 375
EXTENT SIZE clause
in CREATE TABLE statement 350
EXTENT SIZE keywords
in ALTER TABLE statement 149
in CREATE INDEX statement 252
in CREATE TABLE statement 352, 363
Extents
counting how many in a table 1158
default size 363
freeing or reusing in TRUNCATE statement 860
Extents, changing size 1322
External database 1371, 1696
External directives for query optimization 1485
External function
as operator-class support function 270
CREATE FUNCTION 234
dropping 475, 475
executing 502, 628
limits on return values 1272
non-variant 1231
OUT parameter 1153
registering 234
strategy functions 268
variant 1231
EXTERNAL keyword
in CREATE EXTERNAL TABLE statement 201
in External Routine Reference segment 1229
in SAVE EXTERNAL DIRECTIVES statement 685
External language 234
EXTERNAL NAME keywords
in ALTER FUNCTION statement 82
in ALTER PROCEDURE statement 85
in ALTER ROUTINE statement 87
in CREATE FUNCTION statement 225
in CREATE PROCEDURE statement 270
External optimizer directives 685
External procedure
creating body of 279
dropping 480
executing 628
EXTEND role 279
External Routine Reference segment 1229
External routines 1360, 1810
as triggered action 400
CREATE PROCEDURE FROM statement in 281
creating a function in 236
defined 273
dropping 483
EXTEND role 668
pathname syntax 1285
preparing 628
referencing 1229
External synonym 489
External tables 808
adding end-of-line character 220
adding newline character 221
creating 201
loading
from a delimited file 216
tables with the same schema 217
to a fixed text file 216
restrictions in joins and subqueries 717
restrictions on calculating statistics 883
restrictions on optimizer directives 1248
systables data 1373
unloading
to a delimited file 219
to a fixed text file 220
to Informix internal format 220
External users 198, 568
External view 1371
extspace 1327
EXTYPEID field
with DESCRIBE INPUT statement 465
with DESCRIBE statement 461
EXTYPELENGTH field
with DESCRIBE INPUT statement 465
with DESCRIBE statement 461
EXTYPENAME field
with DESCRIBE INPUT statement 465
with DESCRIBE statement 461
EXTYPEOWNERLENGTH field
with DESCRIBE INPUT statement 465
with DESCRIBE statement 461
EXTYPEOWNERNAME field
with DESCRIBE INPUT statement 465
with DESCRIBE statement 461

F

FACT keyword
in optimizer directives 1256
in SET OPTIMIZATION statement 823
FALSE setting
BOOLEAN value 1396
FET_BUF_SIZE environment variable 1483
Fetch buffer 1483
Fetch buffer size 1483
FETCH statement 513, 1506, 1713

ABSOLUTE keyword 1713
description of 1712
sequential 1713
with sequential cursor 1715
Field delimiter
DBDELIMITER 1472
Statements of SQL
LOAD 1472
UNLOAD 1472
Utilities
dbexport 1472
Field literal values
in Literal Row segment 1199
Field of a ROW data type 1432
Field projection 876, 1028, 1584
Field qualifier 1194
EXTEND function 1426
Field, definition of 1583
Fields of a ROW data type 1432
File extensions
.a 1488
.cfg 1494
.cmd 1447
.ec 1461
.ecp 1461
.iem 1473
.jar 1501
.rc 1442, 1445, 1482, 1484
.so 1488
.sql 1385, 1474, 1474, 1481
.std 1503, 1513
FILE keyword
in SET DEBUG FILE statement 797
FILE TO Clause 803
FILE TO keywords
in SET DEBUG FILE statement 798
in SET EXPLAIN statement 800
Files
environment configuration files 1444
for LOAD input 863
installation directory 1497
permission settings 1442
saving output from UNLOAD 863
saving output from UPDATE STATISTICS 895
saving query plans in sqexplain.out 803
sending output with the OUTPUT statement 623
shell 1442
temporary 1477, 1478, 1510
temporary for SE 1478
termcap, terminfo 1500, 1514, 1515
Files, compared to database 1521
FILETOBLOB function 1095, 1096, 1096, 1395
FILETOCLOB function 1095, 1096, 1096, 1398
FILTERING keyword
in ALTER TABLE ADD CONSTRAINT statement 123
in ALTER TABLE statement 108
in CREATE INDEX statement 258
in CREATE TABLE statement 336, 337
in SET CONSTRAINTS statement 782
in SET Database Object Mode statement 784, 788
in SET INDEXES statement 810
Violations table
relationship to diagnostics table 337
with diagnostics tables 855
Filtering mode 1357, 1380, 1685
FILTERING WITH ERROR keywords
in SET Database Object Mode statement 784
FILTERING WITH ERROR NOVALIDATE keywords
in SET Database Object Mode statement 784
FILTERING WITHOUT ERROR keywords
in SET Database Object Mode statement 784
FILTERING WITHOUT ERROR NOVALIDATE keywords
in SET Database Object Mode statement 784
FINAL keyword
in CREATE AGGREGATE statement 185
Finalization function 1326
FIRST clause
description of 1559
using 1560
with ORDER BY clause 1560
FIRST keyword
in ALTER FRAGMENT statement 52, 57
in FETCH statement 513
in OLAP window expressions 1189
in SELECT statement 695, 760
invalid in INSERT 590
FIRST_ROWS keyword
in optimizer directives 1258
in SET OPTIMIZATION statement 820, 822
sysdirectives system catalog table 822
FIXED keyword
in CREATE EXTERNAL TABLE statement 204
Fixed point decimal 1400, 1406
Fixed text files
unloading 216, 220
Fixed-length opaque data type 264
Fixed-length opaque data types 1336
Fixed-length UDT 1383
Fixed-point numbers 1199
Fixed-text files
adding end-of-line character 220
FLOAT data type
built-in casts 1435, 1436
coltype code 1336
defined 1402
display format 1472
literal values 1199
systems not supporting 430
Floating-point decimal 1399, 1402, 1415, 1472
Floating-point numbers 1199
FLOOR function 1057, 1058
FLUSH statement 522
count of rows inserted 1728
rollback 1728
writing rows to buffer 1727
Flushing an insert buffer 640
FOLLOWING keyword in OLAP window expressions 1189
FOR EACH ROW keywords, in CREATE TRIGGER statement 392, 399
for INTERVAL 1194, 1194
FOR keyword 935
in ALTER TRUSTED CONTEXT statement 161
in CONTINUE statement 919
in CREATE FUNCTION statement 225
in CREATE OPCLASS statement 267
in CREATE PROCEDURE statement 270
in CREATE SYNONYM statement 305, 305
in CREATE TRIGGER statement 392, 399
in CREATE TRUSTED CONTEXT statement 413
in DECLARE statement 433, 449
in EXIT statement 929
in GRANT statement 560, 563
in INFO statement 578
in REVOKE statement 673, 675
in SAVE EXTERNAL DIRECTIVES statement 685
in SELECT statement 771, 773
in SET AUTOFREE statement 773
in SET CONSTRAINTS statement 782
in SET Database Object Mode statement 787
in SET INDEXES statement 810
in SET TRIGGERS statement 843
in START VIOLATIONS TABLE statement 844
in STOP VIOLATIONS TABLE statement 856
in UPDATE STATISTICS statement 883, 899
FOR READ ONLY keywords
in DECLARE statement 433
in SELECT statement 773
FOR statement 931
FOR TABLE keywords, in UPDATE STATISTICS statement 883
FOR UPDATE keywords
conflicts with ORDER BY 1725
in DECLARE statement 433
in SELECT statement 690, 771
not needed in ANSI-compliant database 1732
specific columns 1732
with column list 438
FORCE keyword
in UPDATE STATISTICS statement 883, 890
FORCED keyword
in SET OPTIMIZATION statement 823
FOREACH keyword
in CONTINUE statement 919
in EXIT statement 929
FOREACH statement 1776
syntax 935
Foreign key 1682
adding 120
dropping the constraint 130
establishing 109, 331
examples 111, 341
multiple columns 339
validating when enabling 119, 790
Foreign key constraint 111, 123, 341, 670
FOREIGN KEY keywords
in ALTER TABLE statement 117
in CREATE TABLE statement 339, 343
Foreign-key constraint
defining 331
Foreign-key constraints 333
Forest of trees indexes
HASH ON clause 249
FORMAT keyword
in CREATE EXTERNAL TABLE statement 204
FORMAT_UNITS function 1146
Formatting
DATE values with DBDATE 1469
DECIMAL(p) values with DBFLTMASK 1472
FLOAT values with DBFLTMASK 1472
SMALLFLOAT values with DBFLTMASK 1472
Formatting mask
with DBDATE 1469

with DBFLTMASK 1472
FRACTION keyword 1194
FRAGMENT BY clause 1477
FRAGMENT BY keywords
in ALTER FRAGMENT statement 42, 45
in CREATE INDEX statement 255
in CREATE TABLE statement 350, 356
in CREATE TEMP TABLE statement 375
FRAGMENT keyword
in ALTER FRAGMENT statement 23, 42
in ALTER TABLE statement 157
in CREATE TABLE statement 368
in GRANT FRAGMENT statement 573
in REVOKE FRAGMENT statement 679
Fragment-level privilege
granting 573
revoking 679
Fragmentation
adding a fragment 47
adding rowids 98
altering fragments 23
arbitrary rule 358
by expression 35, 52
by list 27, 52, 361
by range interval 27, 52
combining tables 27
Dataskip feature 796
detaching a table fragment 36
distribution strategy 1348
dropping an existing fragment 50, 50
expression 1348, 1469, 1471
Fragment names 63
insufficient log space or disk space 25
list 1348
list of dbspaces 796
modifying an existing fragment
expression 52
nonremainder fragment 52
number of rows in fragment 25
of indexes 255
of tables 356
of temporary tables 375
PDQPRIORITY environment variable 1508
PSORT_NPROCS environment
variable 1512
range interval 26, 31, 38, 67
reinitializing strategy 45
remainder 50
reverting to nonfragmented 42
round robin 1348
round-robin 35
rowid 41
rowid columns with 356
setting priority levels for PDQ 1507
strategy
by expression 358, 797
by round-robin 357, 797
range rule 358
sysfragauth data 1347
sysfragments data 1348
TEXT and BYTE data types 34
transition fragment 63
Fragmentation strategy, modifying 40
FRAGMENTS keyword
in ALTER FRAGMENT statement 52, 57
FRAGMENTS keyword, in INFO statement 578
FREE statement 524
FREE statement, freeing prepared
statements 1720
FROM clause
subqueries in 1638
FROM keyword 1322, 1333
in CREATE ROUTINE FROM statement 284
in DELETE statement 451
in LOAD statement 592
in PREPARE statement 623
in PUT statement 634
in REVOKE FRAGMENT statement 679
in REVOKE statement 654, 671, 673, 675,
678
in SELECT statement 711
in TRIM expressions 1121
FROM keyword, alias names 1577
FULL keyword
in optimizer directives 1249
in SELECT statement 721
Function
companion 503
negator 503
selectivity 1279
Function cursor
defined 935
opening 618
reopening 618
Function expressions 1055
Function keys 1500
FUNCTION keyword
in ALTER FUNCTION statement 82
in DECLARE statement 433
in DROP FUNCTION statement 473
in EXECUTE FUNCTION statement 502
in GRANT statement 546
in REVOKE statement 662
in SELECT statement 720
in UPDATE STATISTICS statement 899
Functional index 241, 245, 245, 260, 1350,
1431, 1480
Functions
aggregate 1592
aggregates of arithmetic expressions 1596
casting 190
collection manipulation 1278
conversion 1600
creating indirectly from a stored file 236
creating with CREATE FUNCTION 225
creating with CREATE FUNCTION
FROM 236
cursor 935
DATE 1600
date-oriented 1596
DBINFO 1613
DECODE 1614
distributed transactions 955
dropping with DROP ROUTINE 481
for BLOB columns 1395
for CLOB columns 1398
for MULTISET columns 1406
in SELECT statements 1591
INITCAP 1606
LOWER 1605
LPAD 1609
modifying
path to executable file 83
modifying routine modifiers 83
name confusion in SPL routine 1771
negator 546
noncursor 935
nonvariant 1231
NVL 1615
protected 473, 482
REPLACE 1606
returned data types 1272
RPAD 1609
smart large object 1095, 1603
specific name 1288
string manipulation 1604
SUBSTR 1608
SUBSTRING 1607
support for complex types 1429
system catalog tables 234
thread-safe 1278
time 1596
TO_CHAR 1601
TO_DATE 1602
trigger 230
unregistering with DROP FUNCTION 473
UPPER 1605
user-defined
defined 273
variant 1231
Functions,
security label support 1092
Functions, data encryption 1617
Functions, SQL
ABS 1058
ACOS 1110
ACOSH 1111
ASCII 1120
ASIN 1111
ASINH 1111
ATAN 1111
ATAN2 1112
ATANH 1111
AVG 1157
CARDINALITY 1070
CASE 1032
CAST 1022
CEIL 1058
CHAR_LENGTH 1092
CHARACTER_LENGTH 1092
CHARINDEXT 1134
CHR 1130
COALESCE 1036
CONCAT 1114
COS 1109
COSH 1109
COUNT 1158
CURRENT 1044
CURRENT_ROLE 1042
CURRENT_USER 1042
CURRVAL 1048
DATE 1102
DAY 1103
DBINFO 1071, 1076
DECODE 1038
DECRYPT_BINARY 1086
DECRYPT_CHAR 1086
DEFAULT_ROLE 1043
ENCRYPT_TDES 1087
Encryption and decryption 1079
EXP 1088
Exponential 1088
EXTEND 1106
FILETOBLOB 1096
FILETOCLOB 1096
FLOOR 1058
FORMAT_UNITS 1146
GETHINT 1088
GREATEST 1058
HEX 1090
INITCAP 1130
INSTR 1136
LEAST 1059

LEFT 1138
LEN 1091
LENGTH 1091
LN 1089
LOCOPY 1099
LOG10 1089
Logarithmic 1088
LOGN 1089
LOTOFILE 1098
LOWER 1130
LPAD 1127
LTRIM 1123
MAX 1162
MDY 1106
MIN 1162
MOD 1060
MONTH 1103
MONTHS_BETWEEN 1104
Natural logarithms 1088
NEXTVAL 1047
NULLIF 1038
NVL 1037
NVL2 1089
OCTET_LENGTH 1092
POW 1060
POWER 1060
QUARTER 1103
RANGE 1163
REPLACE 1126
REVERSE 1125
RIGHT 1139
ROOT 1060
ROUND 1061
RPAD 1128
RTRIM 1124
SECLABEL_BY_COMP 1093
SECLABEL_BY_NAME 1093
SECLABEL_TO_CHAR 1094
SIGN 1095
SIN 1109
SINH 1110
SPACE 1125
SQLCODE 1071
SQRT 1061
STDEV 1163
SUBSTR 1140
SUBSTRB 1141
SUBSTRING 1143
SUBSTRING_INDEXT 1144
SUM 1163
SYS_CONNECT_BY_PATH 746
TAN 1110
TANH 1110
Time 1101
TO_DATE 1107
TO_NUMBER 1107
TODAY 1044
TRIM 1121
TRUNC 1066
UPPER 1130
USER 1042
VARIANCE 1164
WEEKDAY 1104
YEAR 1104
Fuzzy index 269
fwritable gcc option 1493

G

G abbreviation for gigabyte 1146
gcc compiler 1493
Generalized-key index
no renamed table 648
Generic B-trees 1350
Generic CASE expressions 1034
GET DESCRIPTOR statement
use with FETCH statement 518
GET DIAGNOSTICS statement 1346, 1707
SQLSTATE codes 527
syntax 526
GET keyword
in GET DIAGNOSTICS statement 526
GET_TZ option of DBINFO 1078
getenv utility 1441
GETHINT function 1088, 1617
GL_COLLATE table 1373
GL_CTYPE table 1373
GL_DATE environment variable 593, 863, 1205, 1399, 1469, 1469
GL_DATETIME environment variable 584, 592, 593, 863, 1107, 1469
GL_USEGLU environment variable 777
Global environment 921
Global network buffer pool 1488
Global variable
declaring 1772
description of 1772
Global variables 921
GLS environment variables 1445
GNU C compiler 1493
GO TO keywords, in WHENEVER statement 904
GOTO keyword, in WHENEVER statement 907
GOTO statement 940
GRANT FRAGMENT statement 573
GRANT keyword
in GRANT FRAGMENT statement 573
GRANT statement 537, 548, 1365, 1372
GRANT statement, in embedded SQL 1720, 1720
GRANT USAGE ON LANGUAGE statement 1810
Granularity, of locks 1736
Graphic characters 1500
Greater than (>) symbol 1206
Greater than or equal to (>=) relational operator 1549
greaterthan() operator function 1208
greaterthanorequal() operator function 1208
GREATEST function 1057, 1058
GRID keyword of SELECT statement 728
Grid query 728
Grid region 728
Grid table 728
GROUP BY clause 1396, 1415, 1477
description of 1620
GROUP BY keywords
column number with 1620
description of 1620
GROUP BY keywords, in SELECT statement 752
GROUP BY TEXT 1415
Group informix 1473
GROUP keyword
in ALTER USER statement 165
in CREATE DEFAULT USER statement 198
in CREATE USER statement 416
in GRANT statement 568

H

Handle value 1031, 1096
HANDLESNULLS keyword
in CREATE AGGREGATE statement 185
Hash join 804, 1255
HASH keyword
in CREATE INDEX statement 249
HASH ON keywords in CREATE INDEX statement 249
Hash-join 1504
hash() support function 1429
Hashed columns 1348
Hashing parameters 1371
HAVING clause, description of 1620
HAVING keyword 1623
in SELECT statement 754
HCL
Informix
ESQL/C
1461, 1469, 1488, 1505
HEADINGS keyword, in OUTPUT statement 622
Heap size 1502
HEX function 1031, 1090, 1090
HEX function, using in expression 1613
Hexadecimal digits 867, 1472
Hexadecimal dump format 865
Hexadecimal smart-large-object identifier 865, 1098
Hierarchical clause 738
Hierarchical queries
ORDER SIBLINGS BY clause 766
hierarchy
table and row 1589
HIGH INTEG keywords
ALTER TABLE statement 1423
CREATE TABLE statement 1423
in ALTER TABLE statement 149
in CREATE TABLE statement 352
HIGH keyword
in ALTER TABLE statement 149
in SET OPTIMIZATION statement 820
in SET PDQPRIORITY statement 827
in UPDATE STATISTICS statement 883
PDQPRIORITY 1507
UPDATE STATISTICS 1322, 1344
High-availability data replication server (HDR) 374, 817
Histogram 1344
Hold cursor
defined 441
insert cursor with hold 444
update cursor with hold 440
Hold cursor, definition of 1751
HOLD keyword 935
in DECLARE statement 433, 449
Home directory 165, 165, 198, 416, 568
HOME keyword
in ALTER USER statement 165
in CREATE DEFAULT USER statement 198
in CREATE USER statement 416
in GRANT statement 568
Host language 1387
Host variable 1395, 1396, 1415, 1431, 1703
delimiter for 1702
description of 1702
fetching data into 1712
in DELETE statement 1723
in INSERT statement 1727
in UPDATE statement 1731
in WHERE clause 1707
INTO keyword sets 1707
null indicator 1709
restrictions in prepared statement 1718
truncation signalled 1705

hosts.equiv file 181
HOUR keyword 1194
HTML (Hypertext Markup Language) 1398

I

I/O overhead 1512
IBM Data Server Driver for JDBC and SQLJ 911
IBM OpenAdmin Tool (OAT) for Informix 888
ICU libraries 777
Identifier
  column names 1240
  connection name 180
  cursor name 1241, 1244
  defined 1231
  delimited identifiers 1235
  multibyte characters 1233
  non-ASCII characters 1233
  non-unique 1236
  routines 1241
  storage objects 1235
  syntax 1231
  table names 1238, 1240, 1241, 1241
  undelimited identifiers 1233
  uppercase characters 1233
  using keywords 1233
  using keywords as column names 1237
  variable name 1244
IDSLBACREADARRAY keyword
  in GRANT statement 560
  in REVOKE statement 673
IDSLBACREADSET keyword
  in GRANT statement 560
  in REVOKE statement 673
IDSLBACREADTREE keyword
  in GRANT statement 560
  in REVOKE statement 673
IDSLBACRULES
  granting exemptions 561, 674
  in distributed queries 702
IDSLBACRULES keyword
  in CREATE SECURITY POLICY statement 298
IDSLBACWRITEARRAY keyword
  in GRANT statement 560
  in REVOKE statement 673
IDSLBACWRITESET keyword
  in GRANT statement 560
  in REVOKE statement 673
IDSLBACWRITETREE keyword
  in GRANT statement 560
  in REVOKE statement 673
IDSSECURITYLABEL data type 100, 105, 608, 992
  definition 1403
  in distributed queries 561, 674, 702
IDSSECURITYLABEL restrictions 320
IF NOT EXISTS keywords
  in AS SELECT clause of the CREATE TABLE statement 315
IF NOT EXISTSI keywords
  in CREATE DATABASE statement 190
IF statement 941
IF statement, in SPL 1778
IFMXMONGOAUTH environment variable 1483
IFX_ AUTO_REPREPARE session environment variable 26
IFX_ALLOW_NEWLINE function
  effect on quoted strings 1204
  syntax 1148
IFX_AUTO_REPREPARE session environment variable 622, 634, 899
IFX_BIT_BIGSHIFT function 1069
IFX_BIT_LEFTSHIFT function 1069
IFX_BIT_RIGHTSHIFT function 1069, 1070
IFX_DEF_TABLE_LOCKMODE environment variable 136, 137, 366, 1484, 1740
IFX_DIRECTIVES environment variable 1246, 1485
IFX_DIRTY_WAIT environment variable 600, 859
ifx_erkey_1 replication column 347
ifx_erkey_1 shadow column 98
ifx_erkey_2 replication column 347
ifx_erkey_2 shadow column 98
ifx_erkey_3 replication column 347
ifx_erkey_3 shadow column 98
IFX_EXTDIRECTIVES environment variable 686, 1261, 1343, 1485
IFX_EXTEND_ROLE configuration parameter 83, 85, 88, 225, 236, 279, 281, 284, 549, 555, 664, 668, 1285, 1810
ifx_insert_checksum shadow column 98
IFX_LARGE_PAGES environment variable 1486
IFX_LO_SPEC data type 700, 986
IFX_LO_STAT data type 700, 986
IFX_LOB_XFERSIZE environment variable 1487
IFX_LONGID environment variable 1231, 1488
IFX_NETBUF_PVTPOOL_SIZE environment variable 1488
IFX_NETBUF_SIZE environment variable 1489
IFX_NO_SECURITY_CHECK environment variable 1489
IFX_NO_TIMELIMIT_WARNING environment variable 1490
IFX_NODBPROC environment variable 1490
IFX_NOT_STRICT_THOUS_SEP environment variable 1490
IFX_ONTAPE_FILE_PREFIX environment variable 1490
IFX_PAD_VARCHAR environment variable 1491
ifx_replcheck replication column 347
ifx_replcheck shadow column 98
ifx_row_version shadow column 98
IFX_TABLE_LOCKMODE environment variable 451, 602
IFX_UPDDESC environment variable 460, 463, 1491
IFX_XASTDCOMPLIANCE_XAEND environment variable 1492
IFX_XFER_SHMBASE environment variable 1492
IGNORE keyword
  in LAG and LEAD function expressions 1174
  in LAST_VALUE function 1182
Ill-behaved C UDR 1276
IMMEDIATE keyword
  in EXECUTE IMMEDIATE statement 506
  in SET CONSTRAINTS statement 782
  in SET Transaction Mode statement 841
IMPEXP data type 700, 986, 1437
IMPEXPBIN data type 700, 986, 1437
Implicit cast 1332, 1437
Implicit connection 182, 1498
  closing 173
Implicit cursor 935, 1249
IMPLICIT keyword
  in CREATE CAST statement 188
Implicit temporary tables 1477
Implicit transactions 833
Import support function 264, 593
import_binary() support function 1429
import() support function 1429
Importbinary support function 264, 593
in != relational operator 1206, 1206
IN clause 1477
in Collection Subquery segment 965, 965, 965
in Collection-Derived Table segment 1212
in Collection-Subquery segment 965
in CREATE TABLE statement
  ON DELETE CASCADE keywords 111
in DECLARE statement
  WITH HOLD keywords 442, 935
in EXECUTE FUNCTION statement 503, 503
in FOR statement 931, 931, 931, 931
in FOREACH 935
in FOREACH statement 935, 935, 935, 935, 935, 935
in GRANT FRAGMENT statement 574, 574, 574, 574
in GRANT statement 539, 539, 539
in IF statement 942
in INTERVAL field qualifier 1194
in INTERVAL Field Qualifier 1194, 1194, 1194, 1194, 1194, 1194, 1194, 1194, 1194, 1194, 1194, 1194, 1194, 1194
IN keyword 931, 1396, 1406, 1411, 1414, 1439
  as a condition 981
  in ALTER FRAGMENT statement 42, 45, 47, 50, 52, 52
  in ALTER TABLE statement 149
  in Condition segment 972, 981
  in CREATE DATABASE statement 190
  in CREATE FUNCTION statement 225
  in CREATE INDEX statement 251, 254, 255, 361
  in CREATE PROCEDURE statement 270
  in CREATE TABLE statement 350, 352, 356
  in CREATE TEMP TABLE statement 375
  in Data Type segment 993
  in LOCK TABLE statement 598
  in ON EXCEPTION statement 949
  in SELECT statement 733
  to form an intersection 1655
  using in WHERE clause 1547
IN relational operator 1639
in relational operators 1206, 1206, 1206, 1206
in SELECT statement 694
  ALL keyword 574, 728
in SET EXPLAIN output 808
in SET VIOLATIONS TABLE statement
  MAX ROWS keywords 1688
  USING keyword 1688
IN TABLE keywords
  CREATE INDEX statement 1738
IN TABLE keywords, in CREATE INDEX statement 254
IN TABLE storage option 1480
INACTIVE keyword, in SAVE EXTERNAL DIRECTIVES statement 685
INCREMENT keyword
  in ALTER SEQUENCE statement 94
  in CREATE SEQUENCE statement 304
Index
  access method 267
  altering table fragmentation 34
  attached 26, 31, 38, 52, 254, 1348, 1469, 1480, 1512
  B-tree 245, 1350, 1480
  bidirectional traversal 247

BSON field 243
clustered 1350, 1352
clustered fragments 239
composite 245, 339, 1350, 1350
compressed 252
converting during upgrade 883, 902
creating 237
default values for attached 1512
delete flag 889
descending 1350
detached 26, 31, 38, 47, 255, 1480
disabled 259, 793
displaying index information 578
distribution scheme 1480
dropping with ALTER FRAGMENT ONLINE ATTACH 26, 31
dropping with ALTER TABLE . . . DROP CONSTRAINT 476
dropping with DROP INDEX 476
extent size 252
filtering to violations table 793
forest of trees 249, 1480
fragmented 23, 255, 342, 1348
functional 245, 260, 1350, 1431, 1480
fuzzy 269
in-table 260
internal 342
key filter 804
LIST keyword
in CREATE INDEX statement 255
maximum key size 241, 245
multilingual index 778
nonfragmented 42, 1480, 1480
of data types 1389
of environment variables 1449
of system catalog tables 1323
on ORDER BY columns 766
online 260
privilege 23
R-Tree 245, 1480
renaming 645
restrictions on index keys 244
ROOT argument 1060
self-join keys 804
self-join path 1249
shared 106, 111, 127
side-effect 269
sysindexes data 1350
sysindices data 1352
sysobjstate data 1357
system-generated 342, 1247
threads for sorting 1512
unique 258, 1341, 1350, 1412, 1412
restrictions 47
unique keys 1267
virtual 260, 477
INDEX DISABLED keywords
in ALTER TABLE statement 120
INDEX DISABLED keywords in ALTER TABLE statement 117
Index key structure 1352
INDEX keyword
in ALTER FRAGMENT statement 23
in ALTER INDEX statement 84
in CREATE INDEX statement 237
in DROP INDEX statement 476
in GRANT statement 540
in optimizer directives 1249
in RENAME INDEX statement 645
in REVOKE statement 658
INDEX optimizer directive 1249
Index privilege 540, 658, 1372
INDEX_SJ keyword, in optimizer directives 1249
Index-key algorithm 119, 790
INDEXES keyword
in INFO statement 578
in SET Database Object Mode statement 786, 787
in SET INDEXES statement 262, 810
INDEXKEYARRAY data type 700, 986
INDICATOR keyword 503
in EXECUTE statement 498
in FETCH statement 513
in PUT statement 634
in SELECT statement 710
Indicator variable
in expression 1165
Indicator variable, definition of 1709
Indirect typing 928, 1412, 1412
Industry standards, compliance with 1387
INF_ROLE_SEP environment variable 1500, 1500
INFO statement 578
Information Schema views
accessing 1385
columns 1386
defined 1384, 1384
generating 1385
server_info 1388
sql_languages 1387
tables 1386
Informational messages 1346
Informix
database, object-relational databases 1529
Informix
extension checking (DBANSIWARN) 1466
Informix
internal format
unloading 220
INFORMIX keyword
in CREATE EXTERNAL TABLE statement 204
in External Routine Reference segment 1229
informix owner name 1322, 1332, 1344, 1350, 1352, 1373, 1473, 1500
INFORMIX parameter style 1229
informix user name 472, 539, 1225
INFORMIX_SQLCODE keyword, in GET DIAGNOSTICS statement 531
Informix
-Admin group 558
informix.rc file 1442, 1445, 1484
INFORMIXC environment variable 1493
INFORMIXCMCONUNITNAME environment variable 1494
INFORMIXCMNAME environment variable 1493
INFORMIXCONCSMCFG environment variable 1494
INFORMIXCONRETRY environment variable 1495
INFORMIXCONTIME environment variable 1496
INFORMIXCPPMAP environment variable 1497
INFORMIXDIR environment variable 1497
INFORMIXSERVER environment variable 179, 181, 182, 1043, 1492, 1498
INFORMIXSHMBASE environment variable 1498
INFORMIXSTACKSIZE environment variable 1499
INFORMIXTERM environment variable 1500
Inheritance hierarchy 1355, 1410
dropping tables 490
named ROW types 289, 483
INIT keyword
in ALTER FRAGMENT statement 40
in CREATE AGGREGATE statement 185
INITCAP function 1130
INITCAP function, as string manipulation function 1606
Initial-cap characters, converting to 1130
Initialization function 1326, 1365
INOUT parameter
with a statement-local variable 1149
INOUT parameters 1229, 1284
Input support function 264, 1405
input() support function 1429
INSENSITIVE keyword in CREATE DATABASE statement 190
Insert buffer 640
filling with constant values 636
inserting rows
with a cursor 582
storing rows with PUT 634
triggering flushing 640
Insert clause in MERGE statement 602
Insert cursor 440
benefits 440
closing 172
declaring 436
definition of 1727
in INSERT 582
in PUT 636
opening 618
reopening 619
result of CLOSE in SQLCA 172
using 1729
with hold 444
INSERT INTO keywords
in LOAD 598
INSERT keyword 574
in CREATE TRIGGER statement 386
in DECLARE statement 433
in GRANT statement 540
in LOAD statement 592
in MERGE statement 602
in REVOKE FRAGMENT statement 681
in REVOKE statement 658, 658
Insert MERGE operations 1664
Insert privilege 540, 658, 1347, 1372, 1503
INSERT statement 580
INSERT statements 1376, 1380, 1431, 1465, 1469
and end of data 1730
and triggers 400
AT clause 581
collection columns 1668
collection-column values 586
collection-derived table, with 592
constant data with 1729
count of rows inserted 1728
description 1663
effect of transactions 583
embedded 1727
ESQL/C 586, 586, 588

filling insert buffer with PUT 634
in dynamic SQL 592
insert cursor compared with 440
insert triggers 385
inserting
collections 1668
into supertables 1668
multiple rows 1670
rows through a view 582
rows with a cursor 582
into collection cursor 639
lock mode 1747
named row type 1666
null values in collection 1669
nulls 589
number of rows 1704
OUT parameter and SLVs 1153
protected tables 589
row type field values 586
row variables 592
SELECT restrictions 1670
SELECT statement in 1670
selected columns 1665
SERIAL and SERIAL8 columns 585
serial values 1665
smart large objects in 1669
smart large objects with 1031
specifying values to insert 583
unnamed row type 1667
using functions 588
VALUES clause 1663
VALUES clause, expressions with 588
with DECLARE statement 433
with insert cursor 440
with row-type columns 1666
with SELECT statement 590, 1670
Insert trigger 385, 1378
INSERTING operator 230, 975
Inserting rows of constant data 1729
install_jar() procedure 234, 279
Installation directory 1497
INSTEAD OF keywords, in CREATE TRIGGER statement 378
INSTEAD OF trigger 1378, 1831
INSTR function 1136
INT data type 1403
INT8 data type
built-in casts 1435, 1436
coltype code 1336
defined 1403
using with SERIAL8 1395
INTEG keyword 1423
in ALTER TABLE statement 149
in CREATE TABLE statement 352
INTEGER data type
built-in casts 1435, 1436
coltype code 1336
defined 1403
length (syscolumns) 1339
literal values 1198
Intensity attributes 1500
Intent lock 1748
Intent-exclusive locks 26
INTERACTIVE_DESKTOP_OFF environment variable 1501
Internal users 165, 416
INTERNALLENGTH keyword, in CREATE OPAQUE TYPE statement 263
International Components for Unicode (ICU) libraries 777
Internationalized trace messages 1377
Interprocess communications (IPC) 1498
INTERSECT operator
in SELECT statement 690, 756, 759
restrictions on use 756
Intersection
definition of 1655
set operation 1654
INTERVAL data type
as quoted string 1205
coltype code 1336
declaration syntax 995
field qualifier 1194
in expression 1046
in expressions 1424, 1424, 1428, 1429
in INSERT 1205
in relational expressions 1547
length (syscolumns) 1339
loading 593
precision 1194
INTERVAL FIRST keywords
in ALTER FRAGMENT statement 52, 57
Interval fragment 26, 31, 38, 56, 67
INTERVAL keyword
in ALTER FRAGMENT statement 42, 50, 52
in CREATE TABLE statement 356
INTERVAL ONLY keywords
in ALTER FRAGMENT statement 52, 57
INTERVAL TRANSITION keywords in ALTER FRAGMENT statement 52, 63
INTO clause 503, 1713
INTO DESCRIPTOR keywords, in EXECUTE 500
INTO keyword 503, 935
choice of location 1713
in DESCRIBE INPUT statement 463
in DESCRIBE statement 459
in EXECUTE FUNCTION statement 502
in EXECUTE PROCEDURE statement 510, 510
in EXECUTE statement 498
in FETCH statement 513, 1713
in INSERT statement 580
in LOAD statement 592
in MERGE statement 602
in SELECT statement 709
mismatch signalled in SQLWARN 1705
restrictions in INSERT 1670
restrictions in prepared statement 1718
retrieving multiple rows 1711
retrieving single rows 1707
INTO SQL DESCRIPTOR keywords, in EXECUTE statement 500
INTO TEMP clause
invalid in INSERT 590
with UNION operator 756
INTO TEMP keywords, description of 1579
INVISIBLE keyword
in SET INDEXES statement 262
in VISIBLE AND INVISIBLE options for indexes 262
IP address 413
ipcshm protocol 1498
IPCSTR connection 457, 587, 702, 877, 1700
IS keyword
in WHERE clause 734
IS NOT NULL keywords 1552
Condition segment 974
in WHERE clause of a query 734
IS NULL keywords 1552
Condition segment 974
in ALTER FRAGMENT statement 27, 27, 42, 47, 52
in CREATE INDEX statement 361
in CREATE TABLE statement 361
in WHERE clause of a query 734
IS NULL operator 1396
ISAM error code 949, 953, 956, 1704
ISO 8859-1 code set 1388
ISOLATION keyword
in SET ISOLATION statement 812
in SET TRANSACTION statement 837
Isolation level 1388, 1504
ANSI 1743, 1745
Cursor Stability (Informix) 1744
defined 813, 838
description of 1742
dirty read 1743
Informix 1743
read uncommitted 1743
repeatable read 1745
with FETCH statement 519
ISOLATION_LOCKS configuration parameter 1744
Item descriptor 19
ITEM keyword 965
ITEM keyword, collection subquery 1646, 1647
ITER keyword
in CREATE AGGREGATE 185
Iterator functions 720, 956, 1277, 1326
Iterator functions functions 804

J

Jagged rows 343
Jar files
name of a jar ID 1246
renaming 645
Jar management procedures 1501
JAR_TEMP_PATH environment variable 1501
Java class
jar file where defined 1246
package where defined 1287
JAVA keyword
in GRANT statement 549
in REVOKE statement 664
JAVA keyword, in External Routine Reference segment 1229
Java UDRs
access privileges to create 555
CLASS routine modifier 1275
data types of return value 1272
EXTEND role 555
Java signature 1287
shared-object file 1287
static method 1287
Java virtual machine (JVM) 1461, 1501, 1502, 1502
Java Virtual Processor Class
CLASS modifier 1276
Java Virtual-Table Interface 184
JAVA_COMPILER environment variable 1502
JDBC API 1203
JDBC connection 1278
JIT compiler 1502
Join
ANSI outer-join syntax 1629
associative 1574
composite 1568
condition 1568
creating 1570

cross 1571
definition of 1536, 1568
equi-join 1571
hash join 804, 1255
in Condition segment 736
in MERGE statement 602, 1662, 1664, 1671
in UPDATE statement 878
index self-join path 1249
Informix
outer join syntax
1629
left outer 1630
multiple-table join 736, 1254, 1575
natural 1574
nested loop join 1255
nested simple 1632
nested-loop join 804
on derived tables 1630
outer 726, 1628
outer,
Informix
extension syntax
736
right outer 1631
self-join 736, 1625
simple 1568
star-join path 1256
Join filter 602, 724
JOIN keyword
in SELECT statement 721
Join methods 1504
Join operations 1322, 1477
Join-method directive 1255
Join-order directive 1254
JSON data type 700, 986, 987
JVM_MAX_HEAP_SIZE environment variable 1502

K

K abbreviation for kilobyte 1146
KEEP ACCESS TIME keywords
ALTER TABLE statement 1423
CREATE TABLE statement 1423
in ALTER TABLE statement 149
in CREATE TABLE statement 352
KEEP keyword
in ALTER TABLE statement 149
in CREATE TABLE statement 352
keepccomment option of esqlc 1247
Key
primary 1341, 1364, 1364, 1380, 1517
KEY keyword
CREATE TABLE statement 343
in ALTER TABLE statement 106, 117
in CREATE TABLE statement 327, 339
in CREATE TEMP TABLE statement 372, 373
KEY keyword
in CREATE TEMP TABLE statement 373
Key management for encrypted data 1081
Key Only index scan method 804
Key scan 1327
Key-value pair data types
restrictions on index keys 244
Keyboard I/O
INFORMIXTERM setting 1500
TERM setting 1514
TERMCAP setting 1514
TERMINFO setting 1515
keyword MATCHES 1415
Keywords
as identifiers 401, 1233
in a subquery 1639
in a WHERE clause 1547
list for
Informix
1295
Korn shell 1442, 1442

L

Label 1564, 1653
column security label 322
data security label 293, 485
loop label 913
statement label 913, 940
user security label 293, 485, 563, 675
LABEL COMPONENT keywords
in DROP SECURITY statement 485
in RENAME SECURITY statement 647
LABEL keyword
in ALTER SECURITY LABEL COMPONENT statement 88
in CREATE SECURITY LABEL COMPONENT statement 294
in CREATE SECURITY LABEL statement 293
in CREATE SECURITY POLICY statement 298
in DROP SECURITY statement 485
in GRANT statement 563
in RENAME SECURITY statement 647
in REVOKE statement 675
Label-based access control (LBAC) 1403, 1420
LAG function 1174
Language
C 1365, 1461, 1493
C++ 1497
CLIENT_LOCALE setting 1469
DBLANG setting 1473
Extensible Markup Language (XML) 1398
external 234
Hypertext Markup Language (HTML) 1398
Informix
ESQL/C
1423, 1431, 1515
Java 1461, 1501, 1502
privileges on 546, 549, 664
sql_languages information schema view 1387
Stored Procedure Language (SPL) 1431, 1469, 1471
syslangauth data 1356
sysroutinelangs data 1365
LANGUAGE keyword
in External Routine Reference segment 1229
in GRANT statement 546, 549
in REVOKE statement 664
LARGE keyword
in CREATE TABLE statement 319
Large objects 993
constraints 328, 340
distributed storage and staging 357
pointer structure 264
Large pages for virtual memory segments 1486
Large-object data type
defined 1422
listed 1419
LAST COMMITTED keywords, in SET ISOLATION statement 812, 1249
LAST keyword
in FETCH statement 513
in OLAP window expressions 1189
in SELECT statement 760
LAST_DAY function 1101
LAST_VALUE function 1182
LATERAL keyword
in FROM clause of SELECT statement 715
LD_LIBRARY_PATH environment variable 1502
LEAD function 1174
LEADING keyword, in TRIM expressions 1121
Leaf pages 1348
Leap second 1078
LEAST function 1057, 1059
LEFT function 1138
LEFT keyword in SELECT statement 721
Left outer join 1630
LEN function 1090, 1091
LENGTH field
with DESCRIBE INPUT statement 465
with DESCRIBE statement 461
LENGTH function 705, 1090, 1091
on TEXT or BYTE strings 1610
on VARCHAR 1610
use in expression 1610
Less than (<) symbol 1206
Less than or equal to (>=) relational operator 1549
lessthan() operator function 1208
lessthanorequal() operator function 1208
LET statement 943, 1772, 1773
Lettercase conversion 1130
LEVEL keyword
in SELECT statement 746
in SET TRANSACTION statement 837
Level-0 backup 319
libos.a library 1488
LIBPATH environment variable 1503
LIKE 1415
LIKE clause
in SPL function 1769
LIKE keyword
description of 1554
in Condition segment 975
in Routine Parameter List segment 1282
in SELECT statement 734
using in WHERE clause 1547
wildcard characters 734
LIKE keyword of SPL 1412, 1412
LIKE operator 1396, 1439
like() operator function 976
LIMIT Clause 768
LIMIT keyword
in ALTER FRAGMENT statement 52, 57
in SELECT statement 768
LIMIT keywords
invalid in INSERT 590
LIMIT TO keywords
in ALTER FRAGMENT statement 52, 57
Linearized code 1378
List
of data types 1389
of environment variables, by topic 1449
of system catalog tables 1323
LIST data type
coltype code 1336
columns, generating values for 1051
defined 1051
deleting elements from 457
unloading 863
updating elements 882
LIST data type, defined 1404

List fragment clause 361
LIST keyword
in ALTER FRAGMENT statement 42
in CREATE TABLE statement 356
in DEFINE statement 925
in Expression segment 1051
in Literal Collection 1195
LISTING keyword
in CREATE FUNCTION statement 225
in CREATE PROCEDURE statement 270
Literal
DATETIME
in INSERT statement 583
with IN keyword 733
INTERVAL
in expression 1046
in INSERT statement 583
nested row 1202
number 1041
Number 1198
in INSERT 583
with IN keyword 972
opaque data type 1199
ROW 1199
Literal collection
nested 1197
Literal DATETIME
in ALTER TABLE statement 103
Literal number, exponential notation 1199
Literal Row segment 1199
LN function 1089
LO_handles() support function 1429
LOAD statement 592, 1395, 1396, 1415, 1472
Loading data with external tables
data warehouse table
initial load 217
refreshing periodically 218
DELUXE mode 215
express-mode procedure 213
from a delimited file 216
serial columns 217
tables with the same schema 217
to a fixed text file 216
local option to esqlc command 433, 498
Local variable 924
Local variable, description of 1765
Locales
collation order 1373
of trace messages 1377
specifying 1449, 1449
Localized collation 1420
Localized collation order 762, 775, 977, 1209
LOCK keyword
in ALTER TABLE statement 136
in CREATE USER statement 416
in SET LOCK MODE statement 817
LOCK MODE keywords
in ALTER TABLE statement 136
in CREATE TABLE statement 315, 365
Lock table overflow 440
LOCK TABLE statement
syntax 598
LOCK TABLE statement, locking a table explicitly 1737
Lock-table overflow 1484
Locking
and concurrency 1694
behavior of different lock types 1748
blobspaces 39
deadlock 1750
DEF_TABLE_LOCKMODE configuration parameter 365
description of 1735
during
inserts 583
updates 440
effect of FORCE_DDL_EXEC setting 23
end of transaction 1751
exclusive locks 23, 26, 31, 38, 67, 440, 598, 872
granularity 136, 136, 365, 602
IFX_DEF_TABLE_LOCKMODE environment variable 365
in transactions 168
integrity 1734
intent exclusive locks 26, 31, 38, 67
intent locks 1748
intent-exclusive 26
lock duration 1741
number of rows to lock 1744
overriding row-level 601
promotable lock 440
releasing with COMMIT WORK statement 175, 440
releasing with ROLLBACK WORK statement 683
row and key locks 1738
scope of lock 1736
setting lock mode 1749
shared locks 598, 872
time limit 1750
types of locks 365, 1735
coarse index lock 1740
database lock 1736
exclusive 1735
page lock 1738, 1739
promotable 1735
promotable lock 1741
row and key locks 1738
shared 1735
smart-large-object locks 1741
table lock 1737
update cursor 1741
update cursors effect on 440
update lock 1747
update locks 816
WAIT keyword 1749
waiting period 817
when creating a referential constraint 112, 333
with
SET LOCK MODE statement 817
UNLOCK TABLE statement 867
with DELETE 1724
with FETCH statement 519
with SET ISOLATION statement 812
with SET TRANSACTION statement 837
write lock 440
Locking granularity 602
LOCKMODE keyword 1484
LOCKS configuration parameter 472, 601
LOCKS keyword, in SET ISOLATION statement 815
LOCOPY function 1095, 1099, 1395, 1398
LOG keyword
ALTER TABLE statement 1423
CREATE TABLE statement 1423
in ALTER TABLE statement 149
in CREATE DATABASE statement 190
in CREATE TABLE statement 352
in CREATE TEMP TABLE statement 370
in SET LOG statement 819
LOG10 function 1088
Logarithmic functions
LN function 1088
LOG10 function 1088
LOGN function 1089
Logging
buffered versus unbuffered 819
cascading deletes 454
changing mode with SET LOG 819
in CREATE DATABASE statement 190
log space requirements 25
table type options 319
temporary tables 377
with triggers 413
Logging mode 1334
Logical character semantics 320, 928
Logical characters 1420
Logical log
and backups 1693
description of 1692
Logical operator
= (equals) 1553
AND 1553
NOT 1553
OR 1553
Logical operator, in Condition segment 984
LOGN function 1088
Lohandles support function 264
LOLIST data type 700, 986
Long identifiers
client version 1488
IFX_LONGID setting 1488
Information Schema views 1385
Loop
controlled 931
indefinite with WHILE 945, 961
LOOP keyword 931
in CONTINUE statement 919
in EXIT statement 929
LOOP statement 945
Loop, exiting with RAISE exception 1820
LOTOFILE function 1095, 1098, 1098, 1395, 1398
LOW keyword
in SET OPTIMIZATION statement 820
in SET PDQPRIORITY statement 827
in UPDATE STATISTICS statement 883
PDQPRIORITY 1507
UPDATE STATISTICS 1344
LOWER function 1130
LOWER function, as string manipulation function 1605
Lower index filter 804
Lowercase characters, converting to 1130
Lowercase mode codes 1360
Lowercase privilege codes 1317, 1335, 1372
LPAD function 1127, 1127
LPAD function, as string manipulation function 1609
LTRIM function 1123
LVARCHAR data type 986
casting opaque types 1437
coltype code (for client) 1336
defined 1405

M

M abbreviation for megabyte 1146
Machine notes 1500
Machine-independent integer types 1339
Magnetic storage media 1332

mail utility, accessing from an SPL routine 957
Mail, sending from SPL routines 957
Mantissa 1198
Mantissa precision 1386, 1400
Map file for C++ programs 1497
Mapped User 803
MATCHED keyword in MERGE statement 602
MATCHES 1415
MATCHES keyword
in Condition segment 975
in SELECT statement 734
using in WHERE clause 1547
wildcard characters 734
MATCHES operator 1396, 1439
MATCHES relational operator
in WHERE clause 1554
matches() operator function 977
Materialized table expression 714
Materialized view 421
MAX function 1154, 1162
MAX function, as aggregate function 1593
MAX keyword
in ALLOCATE DESCRIPTOR statement 19, 20
in START VIOLATIONS TABLE statement 844
MAX ROWS keywords, in START VIOLATIONS TABLE statement 844
MAX window function 1185
MAX_PDQPRIORITY configuration parameter 827
MAXERRORS keyword
in CREATE EXTERNAL TABLE statement 204
MAXVALUE keyword
in ALTER SEQUENCE statement 94
in CREATE SEQUENCE statement 304
MDY function 1101, 1106
MEDIUM keyword 1322, 1341, 1344
MEDIUM keyword, in UPDATE STATISTICS statement 883
MEDIUM mode 895
Membership (.) operator 1028
Membership operator 1439
Memory
allocating for collection variable 18
allocating for ROW variable 21
deallocating cursors 524
deallocating for collection variable 431
deallocating for cursors 773
deallocating for row variable 433
deallocating prepared objects 524, 775
MERGE statement 602, 1380
using Insert join 1664
using Update join 1671
MERGE statements
using Delete join 1662
Message file
specifying subdirectory with DBLANG 1473
MESSAGE_LENGTH keyword, in GET DIAGNOSTICS statement 531
MESSAGE_TEXT keyword, in GET DIAGNOSTICS statement 531
Messages
chaining 1505
error in syserrors 1346
optimized transfers 1505
reducing requests 1506
trace message template 1377
warning in syserrors 1346
mi_collection_card() function 1404, 1406, 1414
mi_collection* functions 1278
mi_db_error_raise() function 1346
mi_trigger*( ) functions 401
Microsoft C compiler 1493
MIN function 1154, 1162
MIN function, as aggregate function 1593
MIN window function 1185
Minus (-) sign
binary operator 1003
MINUS operator
in SELECT statement 690, 756, 759, 760
restrictions on use 756
minus() operator function 1019
MINUTE keyword 1194, 1194
MINVALUE keyword
in ALTER SEQUENCE statement 95
in CREATE SEQUENCE statement 304
Missing arguments 1210
MITRACE_OFF configuration parameter 1376, 1377
Mixed-case characters, converting to 1130
mkdir utility 1473
MOD function 1057, 1060
MODE ANSI keywords, specifying transactions 1693
MODE keyword
in ALTER TABLE statement 136
in CREATE DATABASE statement 190
in CREATE TABLE statement 315, 365
in LOCK TABLE statement 598
in SET LOCK MODE statement 817
MODERATE INTEG keywords
ALTER TABLE statement 1423
CREATE TABLE statement 1423
in ALTER TABLE statement 149, 352
MODERATE keyword
in ALTER TABLE statement 149
Modifiers
CLASS 1360
COSTFUNC 1360
HANDLESNULLS 1360
INTERNAL 1360
NEGATOR 1360
NOT VARIANT 1360
PARALLELIZABLE 1360
SELCONST 1360
STACK 1360
VARIANT 1360
MODIFY EXTERNAL NAME keywords
in ALTER FUNCTION statement 83
in ALTER PROCEDURE statement 85
in ALTER ROUTINE statement 88
MODIFY keyword
in ALTER ACCESS_METHOD statement 22
in ALTER FRAGMENT statement 52
in ALTER FUNCTION statement 82
in ALTER PROCEDURE statement 85
in ALTER ROUTINE statement 87
in ALTER TABLE statement 139
in ALTER USER statement 165
MODIFY NEXT SIZE keywords 1322
Modifying routine modifiers
with ALTER FUNCTION statement 83
with ALTER PROCEDURE statement 85
with ALTER ROUTINE statement 87
Modulus 1060
MONEY data type
built-in casts 1436
coltype code 1336
defined 1406
in INSERT statement 1664
international money formats 1406
length (syscolumns) 1339
literal values 1199
loading 593
MONTH function 1101, 1103
using, TIME function
MONTH 1597
MONTH function, as time function 1596
MONTH keyword 1194, 1194, 1194
MONTHS_BETWEEN function 1101, 1104
MORE keyword, in GET DIAGNOSTICS statement 530
MQ DataBlade module 1270
Multi-index scan 1249
Multi-index scan path 1249
Multibyte characters
CLOB data type 1398
Multibyte code set 102, 142
Multibyte locales 1092
Multilingual index 778
Multiple triggers
example 387
preventing overriding 411
Multiple-column constraints
in ALTER TABLE statement 117
in CREATE TABLE statement 339
KEY keyword
in CREATE TABLE statement 339
Multiple-table join 1575
Multiple-Table SELECTs 1568
Multiplication sign (*), arithmetic operator 1003
Multirepresentational data 490, 585, 877
Multirow query 516
MULTISET columns, generating values for 1051
MULTISET data type
collection subqueries 965
coltype code 1336
constructor 1431
defined 1051, 1406
deleting elements from 457
unloading 863
updating elements 882
MULTISET keyword 965
collection subquery 1646
in DEFINE statement 925
in Expression segment 1051
in FROM clause of SELECT statement 967
in Literal Collection 1195
Multithreaded application, definition of 1701

N

N setting
sysroleauth.is_grantable 1365
NAME field
with DESCRIBE INPUT statement 465
with DESCRIBE statement 461
NAME keyword
External Routine Reference segment 1229
in ALTER FUNCTION statement 82
in ALTER PROCEDURE statement 85, 85
in ALTER ROUTINE statement 87
in CREATE FUNCTION statement 225
in CREATE PROCEDURE statement 270
Named ROW data type
casting permitted 1438
defined 1409
defining 1409

equivalence 1409
inheritance 1355, 1409
typed tables 1409
Named ROW data types
restrictions on index keys 244
Named row type
assigning with ALTER TABLE 127
associating with a column 999
creating with CREATE ROW TYPE 286
dropping with DROP ROW TYPE 483
inheritance 289
literal values 1199
privileges on 546, 546
Under privilege 670
unloading 863, 866
updating fields 882
Named row type, in VALUES clause 1666
Namer ROW data type
coltype code 1336
Naming convention
database 1224
database objects 1225, 1225
National Language Support (NLS) 1115, 1420
National Language Support (NLS) data types 195
Natural join 1574
NCHAR data type
collation order 1407
coltype code 1336
defined 1407
multibyte characters 1407
NCHAR data type, querying on 1534
negate() operator function 1019
Negator functions 228, 503, 546, 662, 1277, 1360
NEGATOR keyword
Routine Modifier segment 1277
Nested dot notation 1431
Nested loop join 804, 1255
Nested ordering
in SELECT statement 765
Nested ordering, in SELECT 1541
Nested-loop join 1504
NET API 1203
Network buffers 1489
Network environment variable, DBPATH 1474
NEW keyword
in CREATE FUNCTION statement 230
in CREATE PROCEDURE statement 276
in CREATE TRIGGER statement 397
Insert triggers 396
Update triggers 397
in SET USER PASSWORD statement 844
NEW keyword, in CREATE TRIGGER statement 395
Newline character
adding 221
Newline characters in quoted strings 1204
NEXT keyword
in CREATE INDEX statement 252
in CREATE TABLE statement 363
in CREATE TEMP TABLE statement 375
in FETCH statement 513
NEXT SIZE keywords
in CREATE INDEX statement 252
in CREATE TABLE statement 363
NEXT_DAY function 1101
NEXTVAL operator 302, 1046
NFS directory 1478
NLCASE INSENSITIVE database 1117
NLS data types 195
in system catalog tables 1323
NLSCASE database attribute 195
NLSCASE INSENSITIVE database 1132
NLSCASE INSENSITIVE database property 699
NLSCASE INSENSITIVE keywords in CREATE DATABASE statement 190
NLSCASE SENSITIVE keywords in CREATE DATABASE statement 190
NO DEFAULT ROLE keywords
in ALTER TRUSTED CONTEXT statement 161
in CREATE TRUSTED CONTEXT statement 413, 413
NO KEEP ACCESS TIME keywords
ALTER TABLE statement 1423
CREATE TABLE statement 1423
in ALTER TABLE statement 149
in CREATE TABLE statement 352
NO keyword
in ALTER TABLE statement 149
in CREATE TEMP TABLE statement 370
in SET COLLATION statement 775
NO LOG keywords
in ALTER TABLE statement 149
in CREATE TABLE statement 352
in CREATE TEMP TABLE statement 374
no setting of NODEFDAC 1503
NOCACHE keyword
in ALTER SEQUENCE statement 95
in CREATE SEQUENCE statement 305
NOCYCLE keyword
in ALTER SEQUENCE statement 95
in CREATE SEQUENCE statement 304
in SELECT statement 744, 746
NODEFDAC environment variable 228, 229, 274, 275, 544, 662, 1503
effects on new routine 228, 229, 274, 275
effects on new table 345
GRANT statement with 546
NODEFDAC environment variable, effect on privileges of public 1811
NOLOG keyword
ALTER TABLE statement 1423
CREATE TABLE statement 1423
NOMAXVALUE keyword
in ALTER SEQUENCE statement 94
in CREATE SEQUENCE statement 304
NOMINVALUE keyword
in ALTER SEQUENCE statement 95
in CREATE SEQUENCE statement 304
NON_DIM keyword in SET OPTIMIZATION statement 823
Non-default database locales 1323
Non-reserved words 1295
Nonalphanumeric characters 1235
Noncursor function 1274
Nondefault code sets 1209
NONE keyword
in SET ROLE statement 829
NONE role 282, 481
NONE setting
JAVA_COMPILER 1502
Nonexclusive access errors 38
Nonfragmented index 1480
Nonlogging temporary tables
creating 374
duration 377
Nonprintable characters
TEXT data type 1417
VARCHAR data type 1417
Nonvariant functions 1231
NOORDER keyword
in ALTER SEQUENCE statement 95
in CREATE SEQUENCE statement 305
NOT AUTHORIZED keywords
in CREATE SECURITY POLICY statement 298
NOT BETWEEN keywords in WHERE clause 1550
Not equal (!=) relational operator 1549
NOT EXISTS keywords 1656
NOT FOUND keywords, in WHENEVER statement 904
NOT IN keywords 1656
NOT keyword
External Routine Reference segment 1229
in ALTER INDEX statement 84, 85
in BETWEEN condition 971
in Condition segment 967, 972, 976, 982
in MERGE statement 602
in SELECT statement 732, 734
in SET LOCK MODE statement 817
with BETWEEN keyword 733
with IN keyword 735
NOT logical operator 1553
NOT NULL 1415
NOT NULL constraint
collection elements 1404, 1406, 1414, 1431
syscoldepend data 1335
sysconstraints data 1341
NOT NULL keywords 1396, 1404
in ALTER TABLE statement 106, 117
in collection data type declarations 1003
in CREATE ROW TYPE statement 286
in CREATE TABLE statement 327
in CREATE TEMP TABLE statement 372
in DEFINE statement 925
in SELECT statement 732
NOT operator 1439
NOT VARIANT keywords, in External Routine Reference segment 1229
NOT VARIANT routine 1360
NOT WAIT keywords in SET LOCK MODE 817
notequal() operator function 1208
NOVALIDATE keyword
in ALTER TABLE ADD CONSTRAINT statement 123
in ALTER TABLE statement 108
in SET CONSTRAINTS ENABLED statement 782
in SET Database Object Mode statement 784, 788
NTILE function 1180
NULL data type
coltype code 1336
NULL keyword 1209
ambiguous as a routine variable 1242
in ALTER FRAGMENT statement 27, 27, 47, 52
in ALTER TABLE statement 103, 106, 117
in Condition segment 970
in CREATE INDEX statement 361
in CREATE ROW TYPE statement 286
in CREATE TABLE statement 324, 327, 361
in CREATE TEMP TABLE statement 372
in Expression segment 1034, 1035, 1038
in INSERT statement 583
in SELECT statement 732
in SET ROLE statement 829
in UPDATE statement 873, 874
NULL value
allowed or not allowed 1326, 1336

BOOLEAN literal 1396
BYTE data type 1396
Null values
checking for in SELECT statement 498
detecting in ESQL 1709
in IF statement 942
inserting with the VALUES clause 589
invalid for collection types 1003
loading 593
returned implicitly by SPL function 954
testing for 1552
updating a column 874
used in Condition with NOT operator 984
used in the ORDER BY clause 764
WHILE statement 961
with AND and OR keywords 984
with COALESCE function 1036
with logical operator 1553
with NULLIF function 1038
with NVL function 1037
NULLABLE field
with DESCRIBE INPUT statement 465
with DESCRIBE statement 461
NULLIF function 1038
NULLS keyword
in LAG and LEAD function expressions 1174
in LAST_VALUE function 1182
in OLAP window expressions 1189
NULLS keyword, in SELECT statement 760
NUMBER keyword, in GET DIAGNOSTICS statement 530
Numeric data types 992
casting between 1435
casting to character types 1436
listed 1419
NUMROWS keyword
in CREATE EXTERNAL TABLE statement 204
NVARCHAR data type
collation order 1408
coltype code 1336
defined 1408
multibyte characters 1408
NVARCHAR data type, querying on 1534
NVL function 1037, 1615
NVL2 function 1089

O

Object mode
description of 1684
disabled 1685
enabled 1685
filtering 1685
Object mode of database objects 1357
Object-List format, in SET Database Object Mode statement 786
Object-relational database, description of 1529
Object-relational schema 1517
OCTET_LENGTH function 1090, 1092
ODBC API 1203
ODBC driver 1502
OF keyword
in CREATE TRIGGER statement 378, 388, 388
in CREATE VIEW statement 421
in DECLARE statement 433
in DELETE statement 451
in SELECT statement 771
in UPDATE statement 868
OF TYPE keywords
in CREATE TABLE statement 343
in CREATE VIEW statement 421
OFF keyword
in CREATE EXTERNAL TABLE statement 204
in SET EXPLAIN statement 800
in SET PDQPRIORITY statement 827
in SET STATEMENT CACHE statement 833
in TRACE statement 959
OFF setting
IFX_DIRECTIVES 1485, 1485
PDQPRIORITY 1507
OLAP aggregation functions 705
OLAP Aggregation functions 1181
OLAP Numbering function 1171
OLAP Ranking functions 1173
OLAP window expressions 1168
OLAP window functions
aggregation 1168
numbering 1168
Projection clause 706
ranking 1168
syntax 1168
OLD keyword
in CREATE FUNCTION statement 230
in CREATE PROCEDURE statement 276
in CREATE TRIGGER statement
Delete triggers 395
Select triggers 398
Update triggers 397
in SET USER PASSWORD statement 844
OLEDB API 1203
OLTP 476
OLTP (on-line transaction processing) 319
ON DELETE CASCADE keywords 111
in ALTER TABLE statement 109
in CREATE TABLE statement 331
restrictions with triggers 385
ON DELETE CASCADE option 1683
ON EXCEPTION keywords
in Statement Block segment 1289
ON EXCEPTION statement 949
scope of control 1818
trapping errors 1817
user-generated errors 1819
ON keyword
in ALTER FRAGMENT statement 23
in CREATE EXTERNAL TABLE statement 204
in CREATE INDEX statement 237, 249
in CREATE TABLE statement 331, 334
in CREATE TRIGGER statement 380, 386, 387, 388
in GRANT FRAGMENT statement 573
in GRANT statement 545, 546, 549, 550, 560, 567
in MERGE statement 602
in ON EXCEPTION statement 949
in REVOKE FRAGMENT statement 679
in REVOKE statement 658, 661, 662, 664, 665, 673, 678
in SET EXPLAIN statement 800
in SET STATEMENT CACHE statement 833
in TRACE statement 959
ON setting
IFX_DIRECTIVES 1485, 1485
oncheck utility 264, 860, 1090
ONCONFIG environment variable 1503
ONCONFIG parameters
AUTO_REPREPARE 26, 622, 634, 899
AUTO_STAT_MODE 368
DATASKIP 796
DBCREATE_PERMISSION 190
DBSERVERALIASES 176, 457, 587, 702, 877
DBSERVERNAME 457, 587, 702, 877
DEADLOCK_TIMEOUT 819
DEF_TABLE_LOCKMODE 137, 137, 366, 602
DIRECTIVES 1246
DS_MAX_QUERIES 827
DS_NONPDQ_QUERY_MEM 762
EXPLAIN_STAT 804
EXT_DIRECTIVES 686, 1261
IFX_EXTEND_ROLE 668
LOCKS 472, 601
MAX_PDQPRIORITY 827
OPT_GOAL 1258
OPTCOMPIND 834, 1255
PN_STAGEBLOB_THRESHOLD 357
SBSPACENAME 149, 352, 911, 1086, 1087
SQL_LOGICAL_CHAR 928
STACKSIZE 1280
STMT_CACHE 834
STMT_CACHE_HITS 836
STMT_CACHE_NOLIMIT 836
STMT_CACHE_NUMPOOL 833
STMT_CACHE_SIZE 836
SYSSBSPACENAME 157, 368, 890
TEMPTAB_NOLOG 374
UPDATABLE_SECONDARY 817
USELASTCOMMITTED 600, 839
USEOSTIME 1044
USERMAPPING 165, 568
USTLOW_SAMPLE 892
onconfig.std file 1513
ondblog utility 819
oninit command 1484
ONINIT_STDOUT environment variable 1504
ONLINE keyword
CREATE INDEX statement 1738
DROP INDEX statement 1738
in ALTER FRAGMENT statement 26, 26, 31, 38, 67
in CREATE INDEX statement 260
in DROP INDEX statement 476, 477
Online transaction processing 476
Online transaction processing (OLTP) 1348
onload utility 1395, 1396, 1415, 1693
ONLY keyword
in ALTER FRAGMENT statement 52, 57
in DECLARE statement 433
in DELETE statement 451, 453
in SAVE EXTERNAL DIRECTIVES statement 685
in SELECT statement 711, 718
in SET TRANSACTION statement 837
in UPDATE statement 868, 870
in UPDATE STATISTICS statement 883, 893, 895
onmode -Y 803, 804
onmode utility 834
onsecurity utility 1489
onspaces utility 23, 251, 575, 680
onstat utility 796, 1440
onunload utility 1693
onutil utility 1090
Opaque data types
associating with a column 998
cast matrix 1438
comparing 1437
creating 263
dropping 493

in DELETE 457
in DROP TABLE 490
in INSERT 585
in LOAD 597
in UPDATE 877
loading 593, 597
naming 264
storage 1405
support functions 264
sysxtddesc data 1382
sysxtdtypes data 1383
unloading 863
OPAQUE data types
defined 1408
Opaque-type variable 1769
OPEN statement 616, 1506, 1712
Open-Fetch-Close Optimization 800
Opening a cursor 1714
Operator class
btree_ops 270
creating 267
default 270, 1267
default for B-Tree 270
defined 249, 267
dropping with DROP OPCLASS 478
rtree_ops 270
specifying with CREATE INDEX 241, 249
sysams data 1327
sysindices data 1352
sysopclasses data 1358
Operator function
concat() 1021, 1114
divide() 1019
equal() 1208
greaterthan() 1208
greaterthanorequal() 1208
lessthan() 1208
lessthanorequal() 1208
like() 976
matches() 977
minus() 1019
negate() 1019
notequal() 1208
plus() 1019
positive() 1019
times() 1019
operator LIKE 1415
Operator precedence 1439
operator TEXT 1415
OPT_GOAL configuration parameter 1258, 1506
OPT_GOAL environment variable 1258, 1506
OPTCOMPIND configuration parameter 1255, 1504
OPTCOMPIND environment variable 834, 1504
Optim Data Studio 911
Optim Development Studio 911
Optimization
specifying a high or low level 820
OPTIMIZATION keyword
in SET OPTIMIZATION statement 820
Optimizer
and SAVE EXTERNAL DIRECTIVES statement 685
and SET OPTIMIZATION statement 820, 820
Optimizer Directives segment 1246
setting IFX_DIRECTIVES 1485
setting IFX_EXTDIRECTIVES 1485
setting OPT_GOAL 1506
setting OPTCOMPIND 1504
setting OPTOFC 1506
strategy functions 268
with UPDATE STATISTICS 899
Optimizer directives
/BROADCAST 1255
/BUILD 1255
/PROBE 1255
access-method 1249
ALL_ROWS 1258
AVOID_EXECUTE 1259
AVOID_FACT 1256
AVOID_FULL 1249
AVOID_HASH 1255
AVOID_INDEX 1249
AVOID_INDEX_SJ 1249
AVOID_MULTI_INDEX 1249
AVOID_NL 1255
AVOID_STAR_JOIN 1256
AVOID_STMT_CACHE 1261
comment symbols 1247
EXPLAIN 1259
explain-mode 1259
external 686
FACT 1256
FIRST_ROWS 1258
FULL 1249
INDEX 1249
INDEX_ALL 1249
INDEX_SJ 1249
inline 686
join-method 1255
join-order 1254
MULTI_INDEX 1249
negative 1249
optimization-goal 1258
ORDERED 1254
positive 1249
restrictions 1248, 1249, 1256
segment 1246
STAR_JOIN 1256
star-join 1256
statement cache 1261
sysdirectives data 1343
USE_HASH 1255
USE_NL 1255
Optimizer environment settings
FACT 823
NON_DIM 823
Optimizer environment settings
AVOID_FACT 823
STAR_JOIN 823
Optimizing
a database server 820
a query 685, 800
across a network 820
OPTION keyword
in CREATE TRIGGER statement 421
in CREATE VIEW statement 425
in GRANT FRAGMENT statement 573
in GRANT statement 537
OPTION keyword 537
OPTMSG environment variable 1505
OPTOFC environment variable 1506
OR keyword
defined 984
in Condition segment 967
OR logical operator 1553
OR operator 1439
OR relational operator 1551
ORDER 1415
ORDER BY clause 1396, 1477
ORDER BY keywords
ascending order 1540
DESC keyword 1540, 1542
display label with 1567
in OLAP window expressions 1189
in SELECT statement 760
multiple columns 1541
relation to GROUP BY 1620
restrictions in INSERT 1670
restrictions with FOR UPDATE 1725
select columns by number 1542
sorting rows 1539
ORDER BY SIBLINGS keywords
in SELECT statement 766
ORDER keyword
in ALTER SEQUENCE statement 95
in CREATE SEQUENCE statement 305
in OLAP window expressions 1189
ORDERED keyword, in optimizer directives 1254
Ordinal positions 1404
OUT keyword 1281
OUT parameter
default parameter style 1229
user-defined function 1283
with a statement-local variable 1149, 1153
OUTER keyword
in SELECT statement 726
OUTER keyword in SELECT statement 721
Outer-join syntax
ANSI 1629
Informix
1629
OUTPUT keyword
in DESCRIBE statement 459
OUTPUT statement 622
Output support function 264, 1405
output() support function 1429
OVER keyword in OLAP window expressions 1189
Overflow bin 895
Overflow error 1400
Overloaded routine 1210, 1228
OVERRIDE keyword
in CREATE SECURITY POLICY statement 298
Owner 1225
ANSI-compliant database 1264
case-sensitivity 576, 654, 667, 680, 1263
Database Object Name segment 1262
in ANSI-compliant database 576, 654, 667, 680
in CREATE SYNONYM 308
in DROP SEQUENCE 487
in RENAME TABLE statement 648
of a constraint 130
Owner Name segment 1262
qualifier of table names 1238
Owner Name segment 1262
Owner routines 1360, 1503
Owner-privileged UDR 228, 229, 274, 275

P

P abbreviation for page 1146
Package, jar name component 1246
Page footers in sbspaces 1423
Page headers in sbspaces 1423
PAGE keyword
in ALTER TABLE statement 136
in CREATE TABLE statement 365
PAGE lock mode 1373, 1484

Page locking 1738
Page number 1031
Page-level locking
in ALTER TABLE statement 136
in CREATE TABLE statement 365
Parallel database query (PDQ) 1256
Parallel distributed queries
SET PDQPRIORITY statement 827
Parallel distributed queries, setting with PDQPRIORITY 1507
Parallel sorting, setting with PSORT_NPROCS 1510
Parallelizable data query 1278
PARALLELIZABLE keyword, in Routine Modifier segment 1276
Parameter
BYTE or TEXT in SPL 929
dynamic 465
Java method 1287
UDRs 1209
PARAMETER keyword
External Routine Reference segment 1229
Parameterizing a statement, with SQL identifiers 631
Parent table 111, 120
Parent-child relationship 331, 602
Partial-column index 1352
PARTITION BY keywords
in ALTER FRAGMENT statement 42
in CREATE TABLE statement 356
in CREATE TEMP TABLE statement 375
in OLAP window expressions 1189, 1189
PARTITION keyword
in ALTER FRAGMENT statement 27, 36, 42, 45, 45, 47, 52
in CREATE INDEX statement 255, 361
in CREATE TABLE statement 356
in OLAP window expressions 1189
Partition number 1074
Partitions 575, 680
Parts explosion 1716
passwd file 181
PASSWORD keyword
in ALTER USER statement 165
in CREATE USER statement 416
in SET USER PASSWORD statement 844
PATH environment variable 1507, 1507
Pathname
Configuration file
for terminal I/O 1514
for C compiler 1493, 1493
for C++ map file 1497
for concsm.cfg file 1494
for connectivity information 1499
for database server 1474
for dynamic-link libraries 1503
for environment-configuration file 1444
for executable programs 1507
for installation 1497
for message files 1473, 1473
for parallel sorting 1510
for remote shell 1477
for temporary .jar files 1501
for termcap file 1514
for terminfo directory 1515
separator symbols 1507
Pathnames with commas 1097
PB abbreviation for petabyte 1146
PDQ
OPTCOMPIND environment variable 1504
PDQPRIORITY environment variable 1507
SET PDQPRIORITY statement 827
PDQ thread safe functions 1278
PDQPRIORITY environment variable 827, 827, 834
PDQPRIORITY keyword
in SET PDQPRIORITY statement 827
PERCALL_COST keyword, Routine Modifier segment 1275
Percent (%) sign
as wildcard 976
PERCENT_RANK function 1178, 1179
Percentile example 1180
Performance
effect of concurrency 1733
increasing with stored routines 1753
Period
DATE delimiter 1469
Period symbol (.)
DECIMAL values 1199
membership operator 1028
MONEY values 1199
Permissions 1442, 1473
Phantom row 813, 839
Pipe character (|) 204, 592, 867
PIPE keyword
in CREATE EXTERNAL TABLE statement 202
in OUTPUT statement 622
Pluggable authentication module (PAM) 165, 568
Plus (+) sign
arithmetic operator 1003
Plus operator (+)
unary 1198
Plus sign (+)
in optimizer directives 1247
unary operator 1198
plus() operator function 1019
PN_STAGEBLOB_THRESHOLD configuration parameter 357
POINTER data type 700, 986
Pointer to a BYTE or TEXT object 1273
Polar coordinates 1112
POLICY keyword
in ALTER TABLE statement 154
in CREATE SECURITY POLICY statement 298
in CREATE TABLE statement 349
in DROP SECURITY statement 485
in RENAME SECURITY statement 647
positive() operator function 1019
PostScript 1398
POW function 1057, 1060
POWER function 1057, 1060
Precedence in dot notation 1029
Precedence rules
for casts 1437
for lock mode 1484
for SQL operators 1439
for UNIX environment variables 1445
for Windows environment variables 1448
PRECEDING keyword in OLAP window expressions 1189
Precision
of currency values 1406
of numbers 1386, 1399, 1402, 1403, 1403, 1415
of time values 1425, 1428
PRECISION field
with DESCRIBE INPUT statement 465
with DESCRIBE statement 461
PREPARE statement 1373
deferring 799
description of 1718
error return in SQLERRD 1704
for collection variables 626
increasing efficiency 633
multiple SQL statements 1718
multistatement text 507
parameterizing a statement 630
parameterizing for SQL identifiers 631
question (?) mark as placeholder 623
releasing resources with FREE 524
restrictions with SELECT 626
statement identifier 445
statement identifier use 625
syntax 623
valid statement text 626
with external routines 628
with SPL routines 628
Prepared statement 1373
comment symbols in 626
executing 496
prepared object limit 433, 624
setting PDQ priority 827
valid statement text 626
with DESCRIBE INPUT statement 463
with DESCRIBE statement 459
with SPL routines 627
Preserving newline characters in quoted strings 1204
PREVIOUS keyword, in FETCH statement 513
Primary access method 1327, 1371
Primary access methods
modifying 22, 22
Primary key 1341, 1364, 1380, 1412, 1412, 1517
Primary key constraint, definition of 1682
PRIMARY KEY keywords
in ALTER TABLE statement 106, 117
in CREATE TABLE statement 327, 339, 343
in CREATE TEMP TABLE statement 372, 373
Primary key, definition of 1681
PRIMARY keyword
in CREATE ACCESS_METHOD statement 184
Primary thread 1499
Primary-key constraint
cascading deletes 111
data type conversion 143
defining column as 330
dropping 130
requirements for 106, 330
rules of use 331
using 330
printenv utility 1444
Printing with DBPRINT 1476
PRIOR keyword
in SELECT statement 746
PRIOR keyword, in FETCH statement 513
Private environment-configuration file 1444, 1482
PRIVATE keyword
in CREATE SYNONYM statement 305, 305
Private network buffer pool 1488, 1489
Private synonym 1373
Privilege 539
Alter 540
chaining grantors 669
column-specific 660
Connect 539

database-level 656
DBA 539, 656
default table privileges 1503
effect of NODEFDAC 544
Execute 546, 548, 549, 662
for triggered action 408
fragment-level 573
revoking 679
granting 537
in system calls 956
needed to create a cast 188
on a synonym 305
on a view 421
on columns (syscolauth table) 1335
on languages 546, 549, 664
on named row type 546
on procedures and functions (sysprocauth table) 1358
on remote objects 829
on sequences 550, 665
on table fragments 573
on table fragments (sysfragauth table) 1347
on tables (systabauth table) 1372
on the database (sysusers table) 1379
on UDRs called by a trigger 408
on UDTs and named row types (sysxtdtypeauth) 1382
Resource 539
table-level 658
ANSI-compliant 544
column-specific 540
effect on view 544
Usage 549, 662, 664
Privileges
database-level 1678
displaying 1679
needed to modify data 1678
on a database 1678
overview 1524
table-level 1679
PRIVILEGES keyword
in GRANT statement 540
in INFO statement 578
in REVOKE statement 658
Procedural language 1044
Procedure
creating from file 281
DELETING operator 276
dropping with DROP PROCEDURE 479
dropping with DROP ROUTINE 481
external 279
INSERTING operator 276
modifying path to executable file 85
modifying routine modifiers 85
modifying with ALTER PROCEDURE 85
privileges 274, 279
protected 479, 482
SELECTING operator 276
specific name 1288
system catalog tables for 279
trigger 276
UPDATING operator 276
user-defined, definition 273
Procedure cursor
opening 617
PROCEDURE keyword
in ALTER PROCEDURE statement 85
in CREATE PROCEDURE statement 270
in DECLARE statement 433
in DROP PROCEDURE statement 479
in EXECUTE PROCEDURE statement 510
in GRANT statement 546
in REVOKE statement 662
in SELECT statement 720
in TRACE statement 959
in UPDATE STATISTICS statement 899
Procedure-type variables 1769
Program variables
SPL 1702
Projection
clause 694
column with dot notation 1028
field projection 1028
Projection clause 694
Projection list 694, 1028
Projection, definition of 1535
Projects, description of 1529
Promotable lock 440, 1735, 1741
PROPERTIES keyword
in ALTER USER statement 165
in CREATE USER statement 198, 416
in GRANT statement 568
Protected routines 473, 479, 482, 1360
Protected rows 1403, 1420
Protection granularity 154
Pseudo-machine code (p-code) 1359
Pseudo-table 608
Pseudo-users 670
PSM_ACT_LOG environment variable 1508
PSM_CATALOG_PATH environment variable 1508
PSM_DBS_POOL environment variable 1509
PSM_DEBUG environment variable 1509
PSM_DEBUG_LOG environment variable 1510
PSM_LOG_POOL environment variable 1510
PSORT_DBTEMP environment variable 1510
PSORT_NPROCS environment variable 1511
PUBLIC keyword
in ALTER TRUSTED CONTEXT statement 161
in ALTER USER statement 165
in CREATE SYNONYM statement 305, 305
in CREATE TRUSTED CONTEXT statement 413
in GRANT FRAGMENT statement 576
in GRANT statement 552, 567, 568
in REVOKE FRAGMENT statement 680
in REVOKE statement 654, 666, 678
Public synonym 1371, 1373
public user name 1385
Purge policy 52, 57
purge_tables task of the Scheduler 57
Purpose flags
adding and deleting 22
list 1267
Purpose functions 1327
adding, changing, and dropping 22
for access methods 861, 1267
for XA data source types 1270
parallel-execution indicator 1267
Purpose options
valid settings 1267
Purpose values
adding, changing, and dropping 22
PUT clause
in CREATE TABLE statement 350
PUT keyword
in ALTER TABLE statement 149
in CREATE TABLE statement 352
in CREATE TEMP TABLE statement 375
PUT statement
constant data with 1729
count of rows inserted 1728
FLUSH with 634
insert data 1727
sends returned data to buffer 1727
source of row values 635
status code 1728
syntax 634
use in transactions 634
putenv utility 1441

Q

Qualifier field
EXTEND 1428
UNITS 1428
Qualifier, existential 1642
Qualifier, field 1194
Qualifying rows 690
excluded by FIRST option 695
excluded by LIMIT option 695
excluded by SKIP option 695
order of retrieval 697
sort-key order 697
QUARTER function 1101, 1103
Queries
case-insensitive 1117
Query
audit 1642
case-insensitive 1132
compound 1650
cyclic 1684
distributed 700, 702, 986, 996, 1226
execution path 1249, 1256
external databases 1226
optimizer directives 1246
optimizing prepared statements 836
optimizing with SAVE EXTERNAL DIRECTIVES 685
optimizing with SET OPTIMIZATION 820
piping results to another program 623
priority level 827
qualifying rows 690
remote databases 1227
result set 690
self-referencing 1684
sending results to an operating-system file 623
sending results to another program 623
stated in terms of data model 1523
statistics 804
Query optimizer
directives 1485, 1485
external directives 685
recalculating table distributions 883
reoptimizing execution plans 883
sysdistrib data 1344
sysprocplan data 1363
updating distribution data 1322
Query optimizer directive 1246
Query result table 315
Question mark (?)
as placeholder in PREPARE 449, 623
as wildcard 977
dynamic parameters 465
generating unique large-object filename 1098
naming variables in PUT 637
Question Mark (?)
placeholder in PREPARE 619
Quotation marks
delimited identifiers 1235

double 1236
effects of DELIMIDENT environment variable 1235
literal in a quoted string 1205
literal nested collection 1197
owner name 1263
quoted string delimiter 1202, 1205
single 1235
with delimited identifiers 1233
Quoted Pathname segment 1285
Quoted string 1202
as constant expression 1041
DATE and DATETIME literals 1427
DATETIME values as strings 1205
DELIMIDENT setting 1481
effects of DELIMIDENT environment variable 1235
in INSERT 583, 1205
INTERVAL values as strings 1205
invalid with BYTE 1396
LVARCHAR data type 1405
maximum length 1205
newline characters 1148
newline characters in 1204
wildcards 1205
with LIKE keywords 734
Quoted string invalid with TEXT 1415
Quoted string, as constant expression 1664

R

R-tree index 1480, 1512
creating 260
default operator class 270
dropping 477
rtree_ops operator class 270
R-tree secondary-access method 267
Radians
converting degrees to radians 1113
converting radians to degrees 1112
RADIANS function 1108
Radicand 1060
RAISE EXCEPTION statement 953, 1817
Range fragment 26, 31, 38, 56, 67
Range fragmentation 95
RANGE function 1154, 1163
RANGE function, as aggregate function 1594
RANGE keyword
in ALTER FRAGMENT statement 42, 42
in CREATE INDEX statement 255
in CREATE TABLE statement 356, 356
in OLAP window expressions 1189
RANGE window function 1185
RANK function 1177
RATIO_TO_REPORT function 1183
RATIOTOREPORT function 1183
RAW keyword
in ALTER TABLE statement 137
in CREATE TABLE statement 310, 315, 319
RAW table
express-mode loads 213, 217
loading from another database server 219
Re-entrant trigger, description of 1831
Read committed 1388
Read Committed isolation level 839
Read Committed isolation level (ANSI) 1744
READ COMMITTED keywords, in SET TRANSACTION statement 837
READ keyword
in GRANT statement 563
in REVOKE statement 675
in SET ISOLATION statement 813, 814
in SET TRANSACTION statement 837
READ ONLY keywords
in DECLARE statement 433
in SELECT statement 773
in SET TRANSACTION statement 837
Read uncommitted 1388
Read Uncommitted isolation level 838
Read Uncommitted isolation level (ANSI) 1743
READ UNCOMMITTED keywords, in SET TRANSACTION statement 837
READ WRITE keywords, in SET TRANSACTION statement 837
Receive support function 264
RECORDEND keyword 216
in CREATE EXTERNAL TABLE statement 204
Recursive relationship, example of 1716
recv() support function 1429
REFERENCES keyword
in ALTER TABLE ADD CONSTRAINT statement 119
in ALTER TABLE MODIFY statement 119
in ALTER TABLE statement 109
in CREATE FUNCTION statement 1756
in CREATE PROCEDURE statement 1756
in CREATE TABLE statement 331
in EXECUTE FUNCTION statement 502
in EXECUTE PROCEDURE statement 510
in GRANT statement 540
in INFO statement 578
in Return Clause segment 1271
in REVOKE statement 658
REFERENCES keyword, in SPL function 1767
References privilege 1335, 1372
defined 540
displaying 578
revoking 658
REFERENCING keyword
in CREATE FUNCTION statement 230
in CREATE PROCEDURE statement 276
in CREATE TRIGGER statement
Delete triggers 395
Insert triggers 396
Select triggers 398
Update triggers 397
Referential constraint 1341, 1364, 1380
cascading deletes 111
Dataskip feature 797
defining 331
delete triggers 385
dropping 130
locking 333
Referential constraint, definition of 1682
Referential integrity 454
Referential integrity, definition of 1682
REJECTFILE keyword
in CREATE EXTERNAL TABLE statement 204
Relational database, description of 1527
Relational model
join 1536
projection 1535
selection 1535
Relational operation 1534
Relational operators 1206, 1439
BETWEEN 1549
EXISTS 1639
IN 1639
in a WHERE clause 1547
in Condition segment 970
LIKE 1554
MATCHES 1554
NULL 1552
OR 1551
with WHERE keyword in SELECT 732
RELATIVE keyword, in FETCH statement 513
RELEASE keyword
in RELEASE SAVEPOINT statement 641
RELEASE SAVEPOINT statement 641
REMAINDER IN keywords
in ALTER FRAGMENT statement 42, 45, 47, 52
in CREATE INDEX statement 255, 361
in CREATE TABLE statement 356
REMAINDER keyword
in ALTER FRAGMENT statement 27
Remote database 1696
Remote database server 1371, 1483
Remote procedure
restrictions on optimizing 883
Remote query 700, 702
Remote shell 1477
Remote standalone secondary server (RSS) 817
Remote tape devices 1477
RENAME COLUMN statement 643
RENAME CONSTRAINT statement 644
RENAME DATABASE statement 645
RENAME INDEX statement 645
RENAME SECURITY LABEL COMPONENT statement 647
RENAME SECURITY LABEL statement 647
RENAME SECURITY POLICY statement 647
RENAME SECURITY statement 647
RENAME SEQUENCE statement 648, 1516
RENAME TABLE statement 648
RENAME TRUSTED CONTEXT statement 650
RENAME USER statement 653
REOPTIMIZATION keyword in OPEN statement 616
Reoptimizing query plans 885
Repeatable read 1504
Repeatable read isolation level 1745
Repeatable Read isolation level 194, 814, 839
Repeatable Read isolation level, emulating during update 519
REPEATABLE READ keywords
in SET ISOLATION statement 812
in SET TRANSACTION statement 837
REPLACE function 1126
REPLACE function, as string manipulation function 1606
REPLACE keyword
in ALTER TRUSTED CONTEXT statement 161
REPLACE USE FOR keywords
in ALTER TRUSTED CONTEXT statement 161
REPLCHECK keyword
in CREATE TABLE statement 310, 346, 347
Replica identifier 1348
Replication
of data 1694
transparency 1694
REPLICATION keyword
in BEGIN WORK statement 168
Reserved words
delimited identifiers 1233
identifiers 1233
of SQL 1295
RESIDENT configuration parameter 1486

RESOLUTION keyword, in UPDATE STATISTICS statement 895
Resource contention 1507
Resource Grant Manager (RGM) 1348
RESOURCE keyword 539
  in REVOKE statement 656
Resource privilege 539, 1322
  Role
    sysusers data 1379
  System catalog
    authorization identifiers 1379
  with CREATE ACCESS_METHOD statement 184
RESPECT keyword
  in LAG and LEAD function expressions 1174
  in LAST_VALUE function 1182
RESTART keyword, in ALTER SEQUENCE statement 94
RESTRICT keyword
  in CREATE SECURITY POLICY statement 298
  in DROP ACCESS_METHOD statement 469
  in DROP OPCLASS statement 478
  in DROP ROW TYPE statement 483
  in DROP SECURITY statement 485
  in DROP TABLE statement 489
  in DROP TYPE statement 493
  in DROP VIEW statement 494
  in DROP XADATASOURCE statement 495
  in DROP XADATASOURCE TYPE statement 496
  in REVOKE statement 654
RESTRICTED mode of UDRs 831
Restrictions
  external tables 222
Result sets 690, 720
RESUME keyword
  in ON EXCEPTION statement 949
  in RETURN statement 954
RETAIN UPDATE LOCKS keywords
  in SET ISOLATION statement 812
Return Clause segment 1271
RETURN statement 954
Return types, in SPL function 1758
Return value
  declaring in CREATE FUNCTION 1271
  REFERENCES keyword 1273
RETURNED_SQLSTATE field 458, 521
RETURNED_SQLSTATE keyword, in GET DIAGNOSTICS statement 531
RETURNING keyword
  example 235
  in CALL statement 915
  in Return Clause Segment 1271
RETURNS keyword
  in Java Shared-Object-File segment 1287
  in Return Clause segment 1271
REUSE keyword
  in TRUNCATE statement 858
REUSE keyword, in TRUNCATE statement 860
REVERSE function 1125
REVOKE FRAGMENT statement 679
REVOKE statement 549, 654, 1372
REVOKE statement, in embedded SQL 1720, 1720
RIGHT function 1139
RIGHT keyword
  in ANSI Joined Tables segment 721
Right outer join 1631
ROBIN keyword
  in ALTER FRAGMENT statement 42
  in CREATE TABLE statement 356
Role
  activating with SET ROLE 829
  built-in 282, 481, 481
  case-sensitivity 537
  creating with CREATE ROLE 282
  currently enabled 1040
  default 554, 554, 668, 1040
  default role 1379
  default roles 830
  definition 282
  dropping with DROP ROLE statement 481
  enabling with SET ROLE 829
  establishing with CREATE, GRANT, SET 553
  EXTEND 555, 668
  granting privileges with GRANT 554
  granting role with GRANT 553
  INF_ROLE_SEP setting 1500
  revoking privileges 667
  scope of 829
  sysroleauth data 1365
ROLE keyword
  in ALTER TRUSTED CONTEXT statement 161
  in CREATE ROLE statement 282
  in CREATE TRUSTED CONTEXT statement 413
  in DROP ROLE statement 481
  in GRANT statement 554, 567
  in REVOKE statement 654, 668, 678
  in SET ROLE statement 829
Role separation 1500
Roles
  default 1524
  definition 1524
ROLLBACK WORK statement 168, 683
  closes cursors 1751
  releases locks 1741, 1751
  setting SQLCODE 1724
  with WHENEVER 173
ROLLING keyword
  in ALTER FRAGMENT statement 52, 57
Rolling-window fragmentation 1348
root dbspace 190
ROOT function 1057, 1060
ROOT keyword
  in CREATE SECURITY LABEL COMPONENT statement 294
ROUND function 1057, 1061
ROUND ROBIN distributed-storage strategy 357
ROUND ROBIN keywords
  in ALTER FRAGMENT statement 42
  in CREATE TABLE statement 356
Round-robin fragmentation 1348
Rounding error 1209
ROUTINE keyword
  in ALTER ROUTINE statement 87
  in CREATE ROUTINE FROM statement 284
  in DROP ROUTINE statement 481
  in GRANT statement 546, 548, 549
  in REVOKE statement 662
  in UPDATE STATISTICS statement 899
Routine modifier
  CLASS 1275
  COSTFUNC 1276
  HANDLESNULLS 1277
  INTERNAL 1277
  ITERATOR 1277
  NEGATOR 1277
  NOT VARIANT 1280
  PARALLELIZABLE 1278
  PERCALL_COST 1278
  SELCONST 1279
  SELFUNC 1279
  STACK 1280
  VARIANT 1280
Routine Parameter List segment 1281
Routine signature 1228, 1283
Routine statistics 899
Routines
  altering with ALTER ROUTINE 87
  checking references 407
  creating with CREATE ROUTINE FROM 284
  DataBlade API routine 1376
  dropping with DROP ROUTINE 481
  identifier 1360
  modifying
    path to executable file 85
    routine modifiers 87
  modifying path to executable file 88
  overloading 1228
  owner 1360
  privileges 274, 1358
  protected 473, 479, 482, 1360
  restricted 1360
  restrictions in triggered action 407
  specific name 1288
  Stored Procedure Language (SPL) 1431
  syserrors data 1346
  syslangauth data 1356
  sysprocauth data 1358
  sysprocbody data 1359
  sysprocedures data 1360
  sysprocplan data 1363
  sysroutinelangs data 1365
  systraceclasses data 1376
  systracemsgs data 1377
  trigger 230, 276, 1360
ROW constructor, in Expression segment 1049
ROW data types 1432
  casting permitted 1438
  collection-derived tables 1215
  constructor syntax 1049
  dot notation 1028
  dot notation with 1584
  equivalence 1409
  field projection 1584
  field projections in SELECT 1585
  field, definition of 1583
  fields 1330, 1432
  in DELETE statement 1661
  inheritance 1355, 1409
  inserting values 1411
  loading field values 593, 596
  named 1199, 1199, 1409, 1432
  nested 1202
  privileges 546
  selecting columns from 1583
  selecting data from 1581
  selecting fields 708
  selecting from 719
  sysattrtypes data 1330
  sysxtddesc data 1382
  sysxtdtypes data 1382, 1383
  unloading 863, 866
  unnamed 1410, 1432
  updating 876, 882, 1674
  using asterisk notation with SELECT 1585
ROW keyword
  in ALLOCATE ROW statement 21

in ALTER TABLE statement 136
in CREATE ROW TYPE statement 286
in CREATE TABLE statement 365
in CREATE TRIGGER statement 392, 399
in DROP ROW TYPE statement 483
in Expression segment 1049
in Literal Row segment 1199
in OLAP window expressions 1189
ROW lock mode 1373, 1484
Row type columns
definition of 1583
Null values 1675
Row variable
accessing 1223
allocating memory 21
deallocating memory for 433
inserting 586
inserting into 592
selecting from 719
updating 882
ROW_COUNT keyword, in GET DIAGNOSTICS statement 530
ROW_NUMBER window function 1171
Row-column level encryption 1080
Row-level locking
in ALTER TABLE statement 136
in CREATE TABLE statement 365
Row-type columns, generating values for 1050
Row-type data, selecting columns of 1583
Row-type variables, delcaring 1767
ROWID
adding column with INIT clause 41
specifying support 1267
use in a column expression 1031
use in fragmented tables 41
used as column name 1238, 1239
rowid column 41, 356, 765
ROWID keyword, in Expression segment 1025
ROWID, using to locate internal row numbers 1568
ROWIDS 1327
ROWIDS keyword
in ALTER FRAGMENT statement 41
in CREATE TABLE statement 356
ROWNUMBER window function 1171
Rows
checking rows processed in SPL routines 1820
definition of 1528, 1534
deleting 451
finding location 1031
finding number of rows processed 1613
in relational model 1528
inserting 1663
through a view 582
with a cursor 582
locking 1738
number of rows returned 1559
order of qualifying rows 697
phantom row 813
removing 1659
retrieving with FETCH 515
rowid defined 515
uncommitted row 813
updating 1671
updating through a view 870
waiting for a locked row 817
writing buffered rows with FLUSH 522
ROWS keyword in OLAP window expressions 1189
ROWS keyword, in START VIOLATIONS TABLE statement 844
RPAD function 1128
RPAD function, as string manipulation function 1609
RTNPARAMS data type 700
RTNPARAMTYPES data type 986, 1360
RTREE_COST_ADJUST_VALUE environment variable 1512
RTRIM function 1124
RULE keyword
in GRANT statement 560
in REVOKE statement 673
Runtime
warnings (DBANSIWARN) 1466

S

Sample size 1344
SAMPLING keyword, in UPDATE STATISTICS statement 895
SAVE EXTERNAL DIRECTIVES statement 685, 1485
SAVEPOINT keyword
in RELEASE SAVEPOINT statement 641
in ROLLBACK WORK statement 684
in SAVEPOINT statement 688
SAVEPOINT statement 688
Savepoints
destroying 641
setting 688
SBSPACENAME configuration parameter 149, 352, 911, 994, 1086, 1087, 1344
sbspaces
defined 1398, 1423
name 1481
specifying in ALTER TABLE 149
specifying in CREATE TABLE 352
sysams data 1327
syscolattribs data 1334
systabamdata data 1371
SCALE field, with DESCRIBE INPUT statement 465
SCALE field, with DESCRIBE statement 461
Scale of numbers 1386, 1400, 1472
Scan cost 22, 1327
Scheduler
purge_tables task 57
Schema name 1262
Schema Tools 1445
Scope of reference
global 433, 498, 921
in subqueries with UNION 758
local 924
module 433, 498
static 402
Scroll cursors
active set 1715
defined 441
definition of 1713
with FETCH 515
WITH HOLD 816
SCROLL keyword, in DECLARE statement 433
SCROLL keyword, using in DECLARE 1713
SECLABEL_BY_COMP function 1093
SECLABEL_BY_NAME function 1093
SECLABEL_TO_CHAR function 1094
SECOND keyword 1194
Secondary access methods
altering 22
user-defined 22
Secondary data replication server 817
SECONDARY keyword
in CREATE ACCESS_METHOD statement 184
Secondary server 319
Secondary-access methods 1327, 1341, 1352, 1358, 1408
B-tree 267
default operator class 270
defined 237, 267
R-Tree 267
registering 184
Secure auditing 602, 1080
Secure domain name 161
SECURED WITH keywords
in ALTER TABLE statement 113, 144
in CREATE TABLE statement 322
SECURITY keyword
in ALTER SECURITY LABEL COMPONENT statement 88
in ALTER TABLE statement 144, 154
in CREATE SECURITY LABEL COMPONENT statement 294
in CREATE SECURITY LABEL statement 293
in CREATE SECURITY POLICY statement 298
in CREATE TABLE statement 349
in DROP SECURITY statement 485
in GRANT statement 563
in RENAME SECURITY statement 647
in REVOKE statement 675
Security label
assigning to a column 113, 144, 322
creating 293
dropping from a column 144
identifier 1093
in DML operations 602
renaming 485, 647
string format 1094
Security label component
for a security policy 298
renaming 485, 647
SECURITY LABEL COMPONENT keywords
in DROP SECURITY statement 485
in RENAME SECURITY statement 647
Security label components
creating 294
SECURITY LABEL keywords
in ALTER SECURITY LABEL COMPONENT statement 88
in CREATE SECURITY LABEL COMPONENT statement 294
in CREATE SECURITY LABEL statement 293
in CREATE SECURITY POLICY statement 298
in DROP SECURITY statement 485
in GRANT statement 563
in RENAME SECURITY statement 647
in REVOKE statement 675
Security label support functions
SECLABEL_BY_COMP 1093
SECLABEL_BY_NAME 1093
SECLABEL_TO_CHAR 1094
syntax 1092
Security policy 1403
associating with an existing table 154
creating 298
renaming 485, 647
SECURITY POLICY keywords
in ALTER TABLE statement 154
in CREATE SECURITY POLICY statement 298

in CREATE TABLE statement 349
in DROP SECURITY statement 485
in RENAME SECURITY statement 647
Segment
defined 1209
SELCONST keyword routine modifier 1275
Select cursor
declaring 436
opening 618, 1712
reopening 618, 618
using 1711
SELECT INTO clause
no table expressions 714
SELECT INTO TEMP statement 1477
SELECT ITEM keywords, in Collection-Subquery segment 965
SELECT keyword 965
ambiguous use as routine variable 1242
in Condition segment 981, 982, 982
in CREATE TABLE statement 315
in CREATE TRIGGER statement 388
in CREATE VIEW statement 423
in DECLARE statement 433
in GRANT statement 551
in LET statement 943
in OUTPUT statement 622
in REVOKE statement 658, 666
in UNLOAD statement 863
Select list 694
display label 1564
expressions in 1562
functions in 1591, 1613
labels in 1653
selecting all columns 1538
selecting specific columns 1542
specifying a substring in 1546
Select numbers for column names 753
Select privilege 540, 551, 658, 665, 1335, 1372, 1385, 1503
SELECT statements 935, 1322, 1344
accessing collections 1581, 1586
active set 1547, 1707
advanced 1619
aggregate functions in 1154, 1592, 1600
alias names 1577
ALL keyword 1639
and end-of-data return code 1730
ANY keyword 1639
basic concepts 1534
BETWEEN condition 733
collection expressions 1645
collection subquery 1646
collection with 718
collection-derived table 1648
column numbers 765
compound query 1650
cursor for 771, 773, 1711, 1711
date-oriented functions in 1596
description of 1533
display label 1564
DISTINCT keyword 1542
embedded 1707, 1709
executing triggered actions 1829
EXISTS keyword 1642, 1642
FIRST clause 695, 1559
for joined tables 1579
FOR READ ONLY clause 773
for single tables 1538, 1613
FOR UPDATE clause 771
forms of 1533
FROM clause 711
functions 1591, 1613
GROUP BY clause 752, 1620
HAVING clause 754, 1623
IN condition 733
in FOR EACH ROW trigger 394
in INSERT 590
in UPDATE statement 1672, 1672
indicator variables 505
INTO clause with ESQL 709, 1707
INTO TEMP clause 1579
IS NULL condition 734
isolation level 1742
join 1570
joining tables in WHERE clause 736
LIKE or MATCHES condition 734
LIMIT clause 768
multiple-table 1568
natural join 1574
null values in the ORDER BY clause 764
ORDER BY clause 760, 1539
outer join 726, 1628
Projection clause 694
relational-operator condition 732
restrictions in SPL routines 1291
restrictions with INTO clause 626
row type 708, 719
ROWID keyword 1031
select list 1535
select numbers 765
selecting a row type 1581
selecting a substring 1546
selecting expressions 1562
selection list 1538
self-join 1625
set operations 1650
simple 1532
single-table 1538
singleton 709, 1547, 1707
SKIP option 695
smart large objects with 1031
smart-large-object functions in 1603
SPL routine in 707
stand-alone 1829
subquery 1635
subquery with WHERE keyword 732
syntax 690
typed tables in the FROM clause 718
UNION operator 758, 1650
use of expressions 704
user-defined routine in 707
using
for join 1536
for projection 1535
for selection 1535
using functions 1592
with DECLARE 433
with LET 944
writing rows retrieved to an ASCII file 863
Select trigger, description of 1828
Select triggers 388
SELECT triggers 1378
Select, description of 1529
SELECTING operator 230, 975
Selection, description of 1535
Selective row-level auditing 346
Selectivity
functions 1279
of the WHERE clause 895
Selectivity constant 1360, 1360
Selectivity functions 890
Selectivity of an index key 1249
Self-join 1317, 1625
assigning column names with INTO TEMP 1625
defined 736
description of 1625
path 804
with aliases 713
Self-referencing foreign key 119, 790
Self-referencing query 1625, 1684
SELFUNC keyword routine modifier 1275
SELFUNCARGS data type 700, 986, 1280
Semantic integrity 383, 589, 1681
Semicolon (;)
SPL statement block delimiter 279
statement terminator 291
Seminumeric data values 1206
Send support function 264
send() support function 1429
SENDRECV data type 700, 986, 1437
SEQ_CACHE_SIZE configuration parameter 95, 305
Sequence
cache 305
creating a synonym for 305
definition of 1529
dropping a synonym 488
generator 302
privileges on 550, 665
syssequences data 1369
syssynonyms data 1370
syssyntable data 1371
systabauth data 1372
systables data 1373
SEQUENCE keyword
in ALTER SEQUENCE statement 92
in CREATE SEQUENCE statement 302
in DROP SEQUENCE statement 487
in RENAME SEQUENCE statement 648
Sequential cursor
with DECLARE 441
with FETCH 514
Sequential cursor, definition of 1713
Sequential integers
am_id code 1327
classid code 1376
constrid code 1341
extended_id code 1383
langid code 1365
msgid code 1377
opclassid code 1358
planid code 1363
procid code 1360, 1360
seqid code 1369
SERIAL data type 1412
SERIAL8 data type 1412
tabid code 1317, 1369, 1373
SERIAL columns
resetting counter 141
Serial columns, loading 217
SERIAL data type
coltype code 1336
defined 1412
generated number in SQLERRD 1704
inserting a starting value 1665
inserting values 585, 1412
invalid default 324
last inserted value 1077
last SERIAL value inserted 1613
length (syscolumns) 1339
resetting counter 141, 585
resetting values 1412

Serial data types 100
Serial key 797
SERIAL8 data type
  assigning a starting value 1413
  coltype code 1336
  defined 1412
  inserting values 585, 1413
  invalid default 324
  last inserted value 1077
  last SERIAL8 value inserted 1613
  length (syscolumns) 1339
  resetting values 1413
  using with INT8 1395
SERIALIZABLE keyword, in SET TRANSACTION statement 837
Serializable transactions 1388
server_info Information Schema view 1384
SERVER_NAME keyword, in GET DIAGNOSTICS statement 531
Session
  get the client time zone 1078
Session control block 1075
Session ID 1075
Session ID, returned by DBINFO function 1613
SESSION keyword, in SET SESSION AUTHORIZATION statement 831
SET AUTOFREE statement 773
SET clause, in UPDATE statement 1673
SET COLLATION statement 775
SET columns, generating values for 1051
SET CONNECTION statement 535, 779
SET CONSTRAINTS statements 782
SET data type
  coltype code 1336
  defined 1051
  deleting elements from 457
  unloading 863
  updating elements 882
SET data type, defined 1414
SET Database Object Mode statement 1688
  syntax 784
  with
    CREATE TRIGGER statement 411
SET DATASKIP statement 796
SET DEBUG FILE statement
  syntax 797
  with TRACE statement 959
SET DEFERRED_PREPARE statement 799
SET ENCRYPTION PASSWORD statement
  audit-event mnemonic 1080
SET ENVIRONMENT IFX_AUTO_REPREPARE statement 1373
SET ENVIRONMENT statement 1441, 1445, 1504
SET EXPLAIN 803
SET EXPLAIN FILE TO statement 803
SET EXPLAIN ON 804
SET EXPLAIN output 804
SET EXPLAIN statement
  output 808
SET INDEXES statement 262, 810
Set intersection 1655, 1828
SET ISOLATION statement
  and SET TRANSACTION 1742, 1742
  isolation levels defined 838
  similarities to SET TRANSACTION statement 837
  use of 1742
SET keyword
  in ALTER SECURITY LABEL COMPONENT statement 88
  in CREATE SECURITY LABEL COMPONENT statement 294
  in DEFINE statement 925
  in Expression segment 1051
  in Literal Collection 1195
  in MERGE statement 602
  in ON EXCEPTION statement 949
  in SET Database Object Mode statement 784, 784
  in UPDATE statement 873
SET keyword, in MERGE statement 1671
SET keyword, in UPDATE statement 1672
SET LOCK MODE statement 817
SET LOCK MODE statement, description of 1749
SET LOG statement 819
Set operation
  difference 1656
  intersection 1654
  union 1650
  use of 1650
Set operators
  EXCEPT 760
  in distributed queries 756
  MINUS 760
  restrictions on use 756
  UNION ALL 756, 758, 759
SET OPTIMIZATION statement 1506, 1506
  HIGH option 822
  LOW option 822
  syntax 820
SET PDQPRIORITY statement 827, 1507
SET ROLE statement 829
SET SESSION AUTHORIZATION statement 831, 1360
SET STATEMENT CACHE statement 833
SET STMT_CACHE statement 1513, 1513
SET Transaction Mode statement 841
SET TRANSACTION statement 837
  and SET ISOLATION 1742
  default database levels 840
  effects of isolation 840
  similarities to SET ISOLATION statement 837
  use of 1742
SET TRIGGERS statement 843
SET USER PASSWORD statement 844
set utility 1447
Set-column level encryption 1080
setenv utility 179, 1443
Setnet32 1448
setnet32 utility 178
Setnet32 utility 1445
SETSESSIONAUTH access privilege 831
SETSESSIONAUTH keyword
  in GRANT statement 567
  in REVOKE statement 678
SETSESSIONAUTH privilege 831
Setting environment variables
  in UNIX 1441
  in Windows 1445
setUDTextName() procedure 234
setUDTExtName() procedure 279
SGML (Standard Graphic Markup Language) 1398
Shadow columns 95, 98, 98, 346, 347, 348
SHARE keyword, in LOCK TABLE statement 598
Shared class libraries 1529
Shared environment-configuration file 1444
Shared libraries 1488
Shared library functions 555, 1229
Shared lock 1735
Shared lock mode 598
Shared memory
  INFORMIXSHMBASE 1498
Shared-disk secondary server (SDS) 817
Shared-object files 234, 1229
Shell
  remote 1477
  search path 1507
  setting environment variables in a file 1442
  specifying with DBREMOTECMD 1477
Shell script 958
SIBLINGS keyword
  in SELECT statement 766
SIBLINGS keyword, in SELECT statement 760
Side-effect index 269
SIGN function 1095
Signatures 1228
Simple assignment 943
simple large object
  defined 1396
Simple large object data types 993
Simple large objects 993
  defined 1422
  distribution statistics 889
  loading 593, 595
  location (sysblobs) 1332
  unloading 863, 865
Simple large objects, SPL variable 1767
Simple table expression 714
SIN function 1108, 1109
Single quotation marks
  literal in a quoted string 1205
  quoted string delimiter 1202
Single-precision floating-point number 1409, 1415
Single-threaded application 780
Singleton SELECT statement 709, 875, 1547, 1707
SINH function 1110
SITENAME function
  constant expression 1043
  in ALTER TABLE statement 103
  in Condition segment 972
  in CREATE TABLE statement 324
  in DEFINE statement 922
SITENAME function, in INSERT statement 1664
SITENAME function, in SELECT statement 1612
SIZE keyword
  in ALTER TABLE statement 149
  in CREATE EXTERNAL TABLE statement 204
  in CREATE INDEX statement 252
  in CREATE TABLE statement 352, 363
  in CREATE TEMP TABLE statement 375
  in UPDATE STATISTICS statement 895
Size specifications 1146
SKIP keyword in SELECT statement 695
Skip-scan access method 1249
Slash and asterisk (/* */) comment indicator 6, 6, 626, 1247
Slot number 1031
SLV. 1151
SMALL and LARGE
  table type options 319
SMALL keyword
  in CREATE TABLE statement 319
SMALLFLOAT data type

built-in casts 1435, 1436
coltype code 1336
defined 1415
display format 1472
literal values 1199
systems not supporting 430
SMALLINT data type
built-in casts 1435, 1436
coltype code 1336
defined 1415
length (syscolumns) 1339
SMALLINT data type, literal values 1198
Smart large object data types 994
Smart large objects
accessing column data 1031
copying to a file 1098
copying to a smart large object 1099
creating from a file 1095, 1096
data integrity 352
defined 1423
expressions with 1031
extent size 352
functions for copying 1095, 1603
generating filename for 1098
handle values 1031
importing and exporting 1603, 1669
in an UPDATE statement 1677
loading values 593, 596
logging 352
SPL variables 1766
storing 149, 352, 911
syscolattribs data 1334
unloading 863, 865
using SQL functions
in a SELECT statement 1603
in an INSERT statement 1669
SMI. 1075
Solaris operating system 1486
SOME keyword
beginning a subquery 736
in Condition segment 982
SOME keyword, beginning a subquery 1639
SOME operator 1439
Sort-merge join 1504
Sorting
ascending or descending order 241
by CASE expression values 763
DBSPACETEMP environment variable 1477
in a combined query 756
in SELECT 760
nested 1541
PSORT_DBTEMP environment variable 1510
PSORT_NPROCS environment variable 1511
with ORDER BY 1540
SPACE function 1125
Spatial data 490, 877
Spatial queries 1512
Special character, protecting 1558
SPECIFIC FUNCTION keywords
in ALTER FUNCTION statement 82
in GRANT statement 546
in REVOKE statement 662
in UPDATE STATISTICS statement 899
SPECIFIC keyword
EXTERNAL keyword
in ALTER FUNCTION statement 82
Functions
altering with ALTER FUNCTION 82
in ALTER FUNCTION statement 82
in ALTER PROCEDURE statement 85
in ALTER ROUTINE statement 87
in CREATE FUNCTION statement 225
in CREATE PROCEDURE statement 270
in DROP FUNCTION statement 473, 473
in DROP PROCEDURE statement 479
in DROP ROUTINE statement 481
in GRANT statement 546
in REVOKE statement 662
in UPDATE STATISTICS statement 899
Specific Name segment 1288
Specific name, for SPL routine 1756
SPECIFIC PROCEDURE keywords
in ALTER PROCEDURE statement 85
in GRANT statement 546
in REVOKE statement 662
in UPDATE STATISTICS statement 899
SPECIFIC ROUTINE keywords
in ALTER ROUTINE statement 87
in GRANT statement 546
in REVOKE statement 662
in UPDATE STATISTICS statement 899
SPL
assigning values to variables 1772, 1773, 1775
FOREACH loop 1776
LET statement 1772
parameter list 1756
program variable 1702
relation to SQL 1753, 1753
return clause 1758
statement block 1775
tracing triggered actions 1832
using cursors 1776
WITH LISTING IN clause 1761
SPL function
CALL statement 1806
collection query 1792
CREATE FUNCTION 234
cursors 935
data types of return values 1272
dropping 473
dynamic routine-name specification 506, 1808
executing 502, 628
large object variables 1767
optimization 234
registering 234
registering from inside an external routine 236
variant vs. nonvariant 1759
WITH clause 1759
SPL functions
debugging 911
SPL keyword
in GRANT statement 546, 549
in REVOKE statement 664
SPL procedure
creating with CREATE PROCEDURE 279
dynamic routine-name specification 513
executing 628
optimization 279, 885
registering with CREATE PROCEDURE 279
SPL procedures
debugging 911
SPL Routine Debugger 911
SPL routines 1360, 1431, 1469, 1471
adding comments to 1761
as triggered action 400, 1827
BYTE and TEXT data types 929
collection data types 1786
comment indicators 6
comments 1761
compiler messages 1814
CONTINUE statement 1781
debugging 911, 959, 1815
defined 911
definition 273
definition of 1753
dot notation 1785
dropping 1763
dropping with DROP PROCEDURE 479
dynamic routine-name specification 1808
example of 1762
exceptions 1817, 1820
EXECUTE PROCEDURE 1827
executing 1804
executing operating-system commands 956
EXIT statement 1781
exiting a loop 1781
finding errors 1814
FOR loop 1780
handling multiple rows 955
header 920
IF..ELIF..ELSE structure 1778
in an embedded language 1763
in SELECT statement 707
in SELECT statements 1616
introduction to 1753
limits on parameters 1281
name confusion with SQL functions 1770
output file for TRACE statement 797
ownership of created objects 281
passing data 1827
preparing 628
privileges 1809
receiving data from SELECT 709
reoptimizing 885
restrictions when used with DML statements 1293
return types 1758
returning values 1782
row-type data 1785
sending mail 957
setting environment variables 958
simulating errors 953
specific name 1756
SQL expressions 1775
SQL statements not supported 1291
syntax error 1814
system catalog entries 1815
text of 1815
TRACE statement 1832
updating nontriggering columns 1828
uses 1753
variables, scope of 1765
WHILE loop 1780
writing 1754
SPL statements, defined 911
SPL variables 1431
sqexplain.out file 803, 898, 1259
SQL
application languages 1701
Application Programming Interfaces 1701
comments 6
compliance of statements with ANSI standard 14
cursor 1711
description of 1530
dynamic statements 1702
error handling 1709

reserved words 1295
statement types 8
static embedding 1701
SQL (Structured Query Language) 1466
SQL administration API 1146
SQL character set 1481
SQL Communications area
sqlca.sqlcode 1071
SQL Communications Area 172, 1466
altered by end of transaction 1724
description of 1703
inserting rows 1728
result after CLOSE 172
result after DATABASE 430
result after DATASKIP event 796
result after DELETE 458, 1075
result after DESCRIBE 460
result after DESCRIBE INPUT 464
result after FETCH 521
result after FLUSH 523
result after INSERT 382, 1074
result after PUT 640
result after SELECT 1075
result after UPDATE 1075
sqlca.sqlerrd1 1074
sqlca.sqlerrd2 1075
SQL DESCRIPTOR keywords
in DESCRIBE INPUT statement 463
in DESCRIBE statement 459
in EXECUTE statement 498
in FETCH statement 513
in OPEN statement 616
in PUT statement 634
SQL Function. 1058
SQL keyword
in DESCRIBE INPUT statement 465
in DESCRIBE statement 461
in EXECUTE statement 501
in OPEN statement 616
SQL statement cache 1752
disabling 835
enabling 834
prepared statements 836
qualifying criteria 835
SQL statements
restrictions within SPL routines 1291
SQL_INFX_ATTR_DELIMIDENT connection attribute 1235
sql_languages Information Schema view 1384
SQL_LOGICAL_CHAR configuration parameter 102, 142, 320, 1373, 1373, 1420
SQL-99 standard 6
SQLCA. 430, 1077
sqlca.sqlerrd1 382
SQLCODE
end of data 1712
negative values 1710
SQLCODE field
after opening cursor 1712
and FLUSH operation 1728
description of 1703
end of data on SELECT only 1730
end of data signalled 1709
set by DELETE statement 1723
set by PUT statement 1728
SQLCODE function 1071
SQLCODE variable 172, 458, 521, 523, 906
sqld value 518
sqlda structure
in DESCRIBE 459, 460
in DESCRIBE INPUT 463
in EXECUTE 499
in EXECUTE ... INTO 500
in FETCH 518
in OPEN 498, 616, 634, 635
in OPEN...USING DESCRIPTOR 621
SQLERRD array
count of deleted rows 1723
count of inserted rows 1728
count of rows 1730
description of 1704
last inserted BIGSERIAL value 1077
last inserted SERIAL8 value 1077
number of inserted rows 523
syntax of naming 1703
SQLERRM character string 1706
SQLERROR keyword, in WHENEVER statement 904
sqlhosts file 178, 457, 702, 1498, 1499, 1700
SQLHOSTS registry key 702
SQLHOSTS subkey 1499
SQLI client-server communication protocol 527
SQLJ.INSTALL_JAR procedure 1229, 1287
SQLNOTFOUND
with INSERT statement 590
SQLSTATE
after FETCH 521
after FLUSH 523
after REVOKE 660
list of codes 527
not found condition 458, 521, 907
runtime errors 906
warnings 906
SQLSTATE values 1346, 1707
SQLSTATE variable
in non-ANSI-compliant databases 1709
using with a cursor 1712
SQLSTATE, problem values 1710
sqlstypes.h header file 460, 464
sqltypes.h file 1336
sqltypes.h header file 1121
SQLUNKNOWN data type 465
sqlvar structures 518
SQLWARN array 1466
description of 1705
syntax of naming 1703
with PREPARE 1718
SQLWARNING keyword, in WHENEVER statement 904
SQRT function 1057, 1061
SRV_FET_BUF_SIZE environment variable 1512
srvsendrecv data type 264
STABILITY keyword, in SET ISOLATION statement 814
STACK keyword
Routine Modifier segment 1275
Stack size 1360, 1499
STACKSIZE configuration parameter 1280, 1499
Standard deviation, aggregate function 1594
Standard Graphic Markup Language (SGML) 1398
STANDARD keyword
in ALTER TABLE statement 137
in CREATE TABLE statement 310, 315, 319
STANDARD table
DELUXE mode load 215
deluxe-mode load 213
loading data 218
Star join
directives 1256
STAR_JOIN keyword
in optimizer directives 1256
in SET OPTIMIZATION statement 823
Star-join execution plan 823
START DATABASE statement 1474
START keyword
in CREATE SEQUENCE statement 304
in SELECT statement 743
START VIOLATIONS TABLE 1688
START VIOLATIONS TABLE statement 844, 1380
START WITH clause
Hierarchical clause 738
in SELECT statement 743
STAT data type 700, 986, 1344
STATCHANGE configuration parameter 1344
STATCHANGE keyword
in ALTER TABLE statement 157
in CREATE TABLE statement 368
STATCHANGE table attribute 157, 368, 1344
statcollect( ) function 890
Statement block 1775
Statement block segment 1289
Statement cache 1513
Statement cache, SQL 1752
Statement identifier
cursor for 445
defined 625
in DECLARE 433
in FREE 524, 524
in PREPARE 625
releasing 625
STATEMENT keyword, in SET STATEMENT CACHE statement 833
Statement Local Variables
data type of 1151
declaration 1151
expression 1153
INOUT parameter 1149
name space of 1151
OUT parameter 1149, 1283
precedence of 1151
scope of 1153
using 1153
Statements
SQL
ANSI-compliant 14
entering 3
extensions to ANSI standard 15
Statements of SQL
ALTER INDEX 1352
ALTER SEQUENCE 1369, 1516
ALTER TABLE 1322, 1364, 1373, 1516
CLOSE 1506
CONNECT 1474, 1474, 1496, 1498
CREATE ACCESS_METHOD 1327
CREATE AGGREGATE 1326
CREATE CAST 1332, 1436
CREATE DATABASE 1474
CREATE DISTINCT TYPE 1383, 1401, 1518
CREATE FUNCTION 1365, 1503
CREATE IMPLICIT CAST 1518
CREATE INDEX 1317, 1350, 1352, 1373, 1480, 1480
CREATE OPAQUE TYPE 1383, 1408
CREATE OPERATOR CLASS 1358
CREATE PROCEDURE 1359, 1365
CREATE ROLE 1365, 1379
CREATE ROUTINE FROM 1365
CREATE ROW TYPE 1383, 1409

CREATE SCHEMA AUTHORIZATION 1317
CREATE SEQUENCE 1369
CREATE SYNONYM 1371
CREATE TABLE 1342, 1364, 1371
CREATE TRIGGER 1378
CREATE VIEW 1380
CREATE XADATASOURCE 1381
CREATE XADATASOURCETYPE 1381
DATABASE 1474
DECLARE 1506
DELETE 1322, 1363, 1380, 1380
DESCRIBE 1491
DROP CAST 1518
DROP DATABASE 1474
DROP FUNCTION 1360
DROP INDEX 1373
DROP PROCEDURE 1360
DROP ROUTINE 1360
DROP ROW TYPE 1409
DROP SEQUENCE 1516
DROP TABLE 1516
DROP TYPE 1401, 1408
DROP VIEW 1385, 1516
FETCH 1506
GET DIAGNOSTICS 1346
GRANT 1347, 1365, 1372, 1372, 1385
INSERT 1380, 1431, 1465, 1469
LOAD 1396, 1466, 1466
MERGE 1380
OPEN 1506
PREPARE 1373
RENAME SEQUENCE 1516
RENAME TABLE 1516
REVOKE 1372, 1379
SELECT 1322, 1344, 1363, 1477
SET ENVIRONMENT 1504
SET OPTIMIZATION 1506
SET PDQPRIORITY 1507
SET SESSION AUTHORIZATION 1360
SET STMT_CACHE 1513
START DATABASE 1474
START VIOLATIONS TABLE 1380
UNLOAD 1466
UPDATE 1465
UPDATE STATISTICS 1322, 1352, 1479
UPDATE STATISTICS FOR PROCEDURE 1363
UPDATE STATISTICS FOR TABLE 1341
Statements of SQL LOAD 1415
Statements of SQL UPDATE 1415
Statements, SQL
valid only in ESQL/C 1021
static option of ESQL/C 1488
Static SQL 1701
STATIC table
loading data 217
STATISTICS keyword
UPDATE STATISTICS statement 883
STATLEVEL keyword
in ALTER TABLE statement 157
in CREATE TABLE statement 368
STATLEVEL table attribute 157, 368
statprint( ) function 890
STATUS keyword, in INFO statement 578
Status, displaying with INFO statement 578
STDEV function 1154, 1163
STDEV function, as aggregate function 1594
STDEV window function 1185
STEP keyword 931
audit-event mnemonic 1080
STMT_CACHE configuration parameter 834, 1513
STMT_CACHE environment variable 834, 1513
STMT_CACHE keyword 1513
STMT_CACHE_HITS configuration parameter 836, 836
STMT_CACHE_NOLIMIT configuration parameter 836
STMT_CACHE_SIZE configuration parameter 836
STOP keyword, in WHENEVER statement 904
STOP VIOLATIONS TABLE statement 856
STOP_APPLY configuration parameter 817
Storage identifiers 1481
STORAGE keyword
in TRUNCATE statement 858
Storage options, CREATE TEMP TABLE 375
STORE IN keywords
in ALTER FRAGMENT statement 42, 50, 52
STORE keyword
in CREATE TABLE statement 356
Stored Procedure Language 911
Stored procedure language (SPL) 1360, 1431, 1469
Stored routine, general programming 1531
stores_demo database 1516
join columns 1516
Storing smart large objects 352
STRATEGIES keyword, in CREATE OPCLASS statement 268
Strategy functions 268
Stream pipe connection 702, 877, 1700
String-manipulation functions 1113
strings option of gcc 1493
Structured Query Language
UPDATE STATISTICS statement
LOW mode 892
Structured Query Language (SQL) 1466
STYLE keyword
External Routine Reference segment 1229
SUBCLASS_ORIGIN keyword, in GET DIAGNOSTICS statement 531
Subdiagram reference 963
Subquery 965
ALL keyword 1639
ANY keyword 1639
beginning with ALL, ANY, SOME keywords 736
beginning with EXISTS keyword 735, 982
beginning with IN keyword 735, 981
correlated 979, 1635, 1641, 1684
defined 732
estimated cost 804
in Condition segment 979
in DELETE statement 1662
in FROM clause 1638
in select list 1637
in SELECT statement 1635
in UPDATE statement 1672
with SET clause 1672
in WHERE clause 1639
in WHERE clause of UPDATE statement 879
no FIRST keyword 695
single-valued 1640
with DISTINCT keyword 698
with UNION or UNION ALL 758
Subscripting
in a WHERE clause 1558
SPL variables 1770
Subscripting character columns 763, 1030
Subscripts 1396
Subscripts ([ ]), 1415
Subsecond precision 1044
SUBSTR function 1140
SUBSTR function, as string manipulation function 1608
SUBSTRB function 1141
Substring 1546, 1770
in ORDER BY clause of SELECT 763
operator in column expression 1030
SUBSTRING function 1143, 1322
SUBSTRING function, as string manipulation function 1607
Substring functions 1134
SUBSTRING_INDEX function 1144
Subtable 1348, 1355, 1520
excluding from query results 718
inherited properties 344
ONLY keyword in DELETE statement 453
restrictions 290
Subtype 1355, 1409
creating 289
dropping 483
SUM function 1154, 1163
SUM function, as aggregate function 1595
SUM window function 1185
Summary
of data types 1389
of environment variables, by topic 1449
of system catalog tables 1323
superstores_demo database
structure of tables 1517
Supertable 1355, 1520, 1590
in a table hierarchy 1589
inserting into 1668
querying 718
selecting from 1590
using an alias 1591
Supertype 1355, 1409
creating 289
dropping 483
Support functions
assigning 264, 585, 597, 877
comparing 264
defined 270
defining 264
destroy 264, 457, 490
DISTINCT data types 1432
export 264
export( ) 863
exportbinary 264
exportbinary( ) 863
import 264
importbinary 264
input 264
lohandles 264
OPAQUE data types 1408, 1429
output 264
receive 264
routine identifier 1360
send 264
specifying in CREATE OPCLASS 270
SUPPORT keyword, in CREATE OPCLASS statement 267
Surrogate user properties
granting 165, 165, 568
modifying 165
revoking 656
Symbol table 1360, 1360
Synonym
chaining 309, 488
creating 305

difference from alias 305
dropping 488
external 489
syssynonyms data 1370
syssyntable data 1371
systables data 1373
USETABLENAME setting 1516
SYNONYM keyword
in DROP SYNONYM statement 488
syntax 1206
SYS_CONNECT_BY_PATH function 746
sysadmin database 1146
sysaggregates system catalog table 185, 470, 1326
sysams system catalog table 22, 184, 469, 1327
sysattrtypes system catalog table 1330
sysautolocate system catalog table 1331
sysblobs system catalog table 345, 1273, 1332
sysbuiltintypes table 1317
syscasts system catalog table 188, 471, 1022, 1332, 1433
syschecks system catalog table 644, 1333
syscheckudrdep system catalog table 1333
syscolattribs system catalog table 149, 352, 1334
syscolauth system catalog table 286, 1335
syscoldepend system catalog table 1335
syscolumns system catalog table 144, 345, 484, 581, 887, 890, 1336
sysconstraints system catalog table 108, 115, 333, 476, 645, 1341
syscrd database 1317
SYSDATE function
as constant expression 1040
in ALTER TABLE statement 103
in CREATE TABLE statement 324
in DEFINE statement 922
in INSERT statement 588
SYSDATE function, as time function 1596, 1664
sysdbclose
disabling with IFX_NODBPROC 1490
sysdbopen
disabling with IFX_NODBPROC 1490
sysdbopen() procedure 827
sysdefaults system catalog table 1342
sysdepend system catalog table 1343
sysdirectives system catalog table 685, 688, 1261, 1343
sysdistrib system catalog table 261, 887, 890, 1344
sysdomains system catalog view 1346
sysecpolicies system catalog table 298, 485, 647
syserrors system catalog table 1346
sysextcols system catalog table 345
sysextdfiles system catalog table 345
sysexternal system catalog table 345
sysfragauth system catalog table 282, 574, 577, 1347
sysfragdist system catalog table 157
sysfragments system catalog table 25, 26, 31, 35, 120, 345, 645, 1074, 1348
sysindexes system catalog table 336, 476, 645, 1350
sysindexes system catalog tables 1352
sysindices system catalog table 252, 336, 887
sysinherits system catalog table 345, 484, 1355
syslangauth system catalog table 1356
syslogmap system catalog table 1356
sysmaster database 345, 1075, 1158, 1317
contrasted with system catalog tables 1317
initialization 1440
sysobjstate system catalog table 108, 120, 123, 333, 337, 645, 784, 784, 792, 793, 1357
sysopclasses system catalog table 1358
sysprocauth system catalog table 234, 279, 282, 345, 1358
sysprocbody system catalog table 234, 278, 278, 1081, 1359
sysprocbody, system catalog table 1815
sysproccolumns system catalog table 1360
sysprocedures system catalog table 234, 279, 479, 482, 831, 1360
sysprocplan system catalog table 234, 279, 899, 1363
SYSPURGE( ) function 57
sysreferences system catalog table 1364
Sysrem catalog tables
sysconstraintst 115
sysroleauth system catalog table 282, 1365
sysroutinelangs system catalog table 1365
SYSSBSPACENAME configuration parameter 157, 368, 883
SYSSBSPACENAME onconfig parameter 890
sysseclabelauth system catalog table 565, 677, 1366
sysseclabelcomponentelements system catalog table 296, 1366
sysseclabelcomponents system catalog table 88, 296, 485, 647, 1366
sysseclabelnames system catalog table 1367
sysseclabels system catalog table 144, 293, 485, 647, 1367
syssecpolicies system catalog table 1367
syssecpolicycomponentrules system catalog table 298
syssecpolicycomponents system catalog table 1368
syssecpolicyexemptions system catalog table 1368
syssequences system catalog table 92, 94, 302, 487, 1369
syssurrogateauth system catalog table 1370
syssynonyms system catalog table 488, 1370
syssyntable system catalog table 488, 1371
systabamdata system catalog table 1371
systabauth system catalog table 282, 289, 574, 1372
systables system catalog table 144, 345, 476, 488, 661, 887, 892, 1074, 1373
System administrator (DBA) 1317
System applet 1446
SYSTEM AUTHID keywords
in ALTER TRUSTED CONTEXT statement 161
in CREATE TRUSTED CONTEXT statement 413
System catalog
access methods 1327, 1371
access privileges 1335, 1347
accessing 1322
altering contents 1322
casts 1332
columns 1336
complex data types 1330, 1383
constraint violations 1380
constraints 1333, 1335, 1341
data distributions 1344
database tables 1373
default values 1342
defined 1316
dependencies 1343
discretionary access privileges 1372
drvurity policies 1367
example 1317
external directives 1343
fragment privileges 1347
fragments 1348
indexes 1350, 1352
inheritance 1355
list of tables 1323
messages 1346
operator classes 1358
privileges 1379, 1382
programming languages 1356, 1365
referential constraints 1341, 1364, 1380
roles 1365
routine parameters 1360
routines 1358, 1360, 1363
security label components 1366
sequence objects 1369
simple large objects 1332
smart large objects 1334
synonyms 1370
text of routines 1359
trace classes 1376
trace messages 1377
triggers 1378, 1378
updating 1322
use by database server 1317
user-defined aggregates 1326
user-defined data types 1382, 1383
views 1373, 1380
XA data source types 1381
XA data sources 1381
System catalog table
SYSCOLAUTH 286
SYSXTDTYPES 286
System catalog tables
owner informix 1225
synonyms 1371
sysaggregates 185, 1326
sysams 22, 184, 184, 1327
sysams s 22
sysattrtypes 1330
sysautolocate 1331
sysblobs 1273, 1332
syscasts 188, 471, 1022, 1332
syschecks 644, 1333
syscheckudrdep 1333
syscolattribs 149, 1334
syscolauth 282, 1335
syscoldepend 1335
syscolumns 484, 887, 1336
sysconstraints 108, 333, 336, 476, 645, 1341
sysdefaults 1342
sysdepend 494, 1343
sysdirectives 685, 1261, 1343
sysdistrib 261, 887, 1344
sysdomains 1346
sysecpolicies 298, 485, 647
syserrors 1346
sysextcols 345
sysextdfiles 345
sysexternal 345
sysfragauth 282, 574, 577, 1347
sysfragdist 887

sysfragments 25, 26, 31, 120, 345, 645, 1348
sysfragmentst 887
sysindexes 336, 645, 887, 1350
sysindices 252, 336, 1352
sysinherits 484, 1355
syslangauth 1356
syslogmap 1356
sysobjstate 108, 120, 123, 333, 645, 645, 784, 793, 1357
sysopclasses 1358
sysprocauth 234, 279, 282, 1358
sysprocbody 234, 279, 1081, 1359
sysproccolumns 1360
sysprocedures 234, 279, 479, 482, 831, 1360
sysprocplan 234, 279, 899, 1363
sysreferences 1364
sysroleauth 282, 1365
sysroutinelangs 1365
syssecLabelauth 565, 677, 1366
syssecLabelcomponentelements 88, 296, 1366
syssecLabelcomponents 88, 296, 485, 647, 1366
syssecLabelnames 1367
syssecLabels 293, 485, 647, 1367
syssecpolicies 1367
syssecpolicycomponentrules 298
syssecpolicycomponents 1368
syssecpolicyexemptions 1368
syssequences 92, 94, 302, 487, 1369
syssurrogateauth 1370
syssynonyms 488, 1370
syssyntable 488, 1371
systabamdata 1371
systabauth 282, 289, 557, 574, 1372
systables 476, 484, 488, 661, 887, 1373
systraceclasses 1376
systracemsgs 1377
systrigbody 1378
systriggers 383, 491, 1378
sysusers 282, 290, 1379
sysviews 421, 644, 648, 1380
sysviolations 844, 857, 1380
sysxadatasources 426, 495, 1381
sysxasourcetypes 428, 496, 1381
sysxtddesc 1382
sysxtdtypeauth 263, 282, 289, 345, 545, 1382
sysxtdtypes 263, 289, 483, 545, 1383
System catalogs
creating 190
dropping tables 491
privileges in 1679
querying 1679
sysprocbody 1815
systabauth 1679
System clock 1040
System constants 1040
System descriptor area 1719
System index 26, 31, 38, 257, 1247
System name
database qualifier 1225
SYSTEM statement 956
System-descriptor area
creating 19
deallocating 432
item descriptors 19
OPEN using 620, 638
use with EXECUTE statement 501
with ALLOCATE DESCRIPTOR 19
with DESCRIBE 461
with DESCRIBE INPUT 465
with EXECUTE ... INTO 500
System-diagnostics area 521
System-monitoring interface tables 862, 1075, 1158
SYSTEM() command, on NT 1501
systraceclasses system catalog table 1376
systracemsgs system catalog table 1377
systrigbody system catalog table 1378
systriggers system catalog table 383, 491, 1378
sysusers database tables
systcxattributes 161, 413, 492, 650
systcxusers 161, 413, 492, 650
systrustedcontext 161, 413, 492, 650
sysusers system catalog table 282, 290, 345, 578, 1379
sysutils database 1317
sysuuid database 1317
sysviews system catalog table 421, 644, 648, 1380
sysviolations system catalog table 844, 857, 1380
sysxadatasources system catalog table 426, 495, 1381
sysxasourcetypes system catalog table 428, 496, 1381
sysxtddesc system catalog table 1382
sysxtdtypeauth system catalog table 200, 263, 282, 289, 345, 545, 1382
sysxtdtypes system catalog table 200, 263, 286, 289, 290, 345, 483, 493, 545, 1383, 1408, 1409

T

T abbreviation for terabyte 1146
tabid 1317, 1373
tabid column 40
Table
adding a constraint 115, 117
adding a constraint to a column with data 148
alias in DELETE 456
alias in SELECT 711
alias in UPDATE 872
build 804
changing a column data type 1433
child 331
compressed 365
consumed 26, 31
creating 310
creating a synonym for 305
default privileges 574
defining fragmentation strategy 356
deleting all rows 858
dependencies, in sysdepend 1343
derived 714, 967, 1214
description of 1528
diagnostic 1380
diagnostics 382, 602, 853
diagnostics table 856
dropping 489
dropping a synonym 488
dropping a system table 491
extent size 1373
external 201, 201, 222, 382, 1248
fragmented 23, 1348
hash 804
hashing parameters 1371
hierarchy 1348, 1355, 1409, 1520, 1589
in relational model 1528
inheritance 383, 391
inheritance hierarchy 343, 870
inheritance, sysinherits data 1355
isolating 351
joins in Condition segment 736
loading data
with onload utility 1693
loading data with the LOAD statement 592
lock 1737
lock mode 1373, 1484
locking
with ALTER INDEX 84
with LOCK TABLE 598
nonfragmented 42, 1480
nonlogging 374
not in the current database 1553
operations on a 1529
parent 331
permanent 376
privileges
granting 540
privileges on 345, 658
protected 40, 589
qualifiers 1227
query result table 315
raw 871
renaming 648
separate from large object storage 1422
static 838
structure in superstores_demo database 1517
surviving 26, 31
synonyms in syssyntable 1370
systables data 1373
system catalog 382
system catalog tables 1326
target 844, 848, 856
temporary 370, 382, 885, 1477, 1478
temporary in SE 1478
temporary table name 371
typed 286, 344, 888
unlocking 867
untyped 344
untyped, and unnamed ROW 1411
updating statistics 885
version value 1373
violations 382, 602, 847, 1380
violations table 856
virtual 1212
waiting for a locked table 817
Table expression 714, 967, 1214
Table format, in SET Database Object Mode statement 787
Table hierarchy
triggers in 1828
UPDATE statements 1676
TABLE keyword 1212
in ALTER FRAGMENT statement 23
in ALTER TABLE statement 136, 157
in CREATE EXTERNAL TABLE statement 201
in CREATE INDEX statement 254
in CREATE TABLE statement 368
in CREATE TEMP TABLE statement 370
in Data Type segment 993
in DROP TABLE statement 489
in LOCK TABLE statement 598
in RENAME TABLE statement 648
in SELECT statement 720

in START VIOLATIONS TABLE statement 844
in STOP VIOLATIONS TABLE statement 856
in TRUNCATE statement 858
in UNLOCK TABLE statement 867
in UPDATE STATISTICS statement 883
TABLE_SIZE
in SET ENVIRONMENT statement 377
TABLE_SIZE environment variable 377
Table-based fragmentation 1348
Table-level privileges 556
PUBLIC 1385
sysfragauth data 1347
systabauth data 1317, 1372
tables Information Schema view 1384
TABLES keyword, in INFO statement 578
TAN function 1108, 1110
TANH function 1110
Tape management
setting DBREMOTECMD 1477
Target table
relationship to diagnostics table 848, 856
relationship to violations table 848, 856
TASK function 1146
TCP/IP connection 457, 587, 702, 702, 877, 1700
TDES (Triple Data Encryption Standard) 1087
TEMP keyword
in CREATE TEMP TABLE statement 370
Temporary dbspace 375, 1477
Temporary files 1478
in SE, specifying directory with DBTEMP 1478
setting DBSPACETEMP 1477
setting PSORT_DBTEMP 1510
Temporary tables 1477
and active set of cursor 1714
and fragmentation 375
assigning column names 1625
constraints allowed 372
creating 370
creating constraints for 372
defining columns 371
differences from permanent tables 376
duration 377
example 1670
in SE, specifying directory with DBTEMP 1478
INFO statement restrictions 376
logging 374
specifying dbspace with DBSPACETEMP 1477
storage 375
updating statistics 885
when deleted 377
TEMPTAB_NOLOG configuration parameter 374
Tenant databases 149
TERM environment variable 1514
TERMCAP environment variable 1514
termcap file
setting INFORMIXTERM 1500
setting TERMCAP 1514
Terminal handling
setting INFORMIXTERM 1500
setting TERM 1514
setting TERMCAP 1514
setting TERMINFO 1515
terminfo directory 1500, 1515
TERMINFO environment variable 1515
TEST keyword, in SAVE EXTERNAL DIRECTIVES statement 685
TEXT 1415
TEXT argument 1415
TEXT Character string TEXT 1415
TEXT column
changing the data type 141
distribution statistics 889
TEXT data type 1415, 1415
coltype code 1336
declaration syntax 993
increasing buffer size 1466
length (syscolumns) 1339
loading 593
nonprintable characters 1417
restrictions
with GROUP BY 1620
setting buffer size 1466
SPL routines 921, 929
storage location 993
sysblobs data 1332
sysfragments data 1348
unloading 863, 865
using LENGTH function on 1610
with control characters 1417
with relational expressions 1547
TEXT data type IS NULL 1415
TEXT data type restrictions 1415
Text editor 1472
TEXT keyword
in Data Type segment 993
in Return Clause segment 1271
THEN keyword
in CASE statement 916
in Expression segment 1034, 1035
in IF statement 941
in MERGE statement 602
thousands separator 1490
Thousands separator 1406
thread flag of ESQL/C 1515
Thread-safe application
defined 469, 780, 782
THREADLIB environment variable 1515
Time and date, getting current 1044
Time data types 995
arithmetic 1424
length (syscolumns) 1339
listed 1419
Time function
description of 1596
restrictions with GROUP BY 753
use in Function Expressions 1101
use in SELECT 705, 1592
TIME function
DAY and CURRENT 1596
WEEKDAY 1598
YEAR 1599
TIME keyword
in ALTER TABLE statement 149, 149
in CREATE TABLE statement 352
Time unit
INTERVAL data types 1194
Time unitMONTHS_BETWEEN expressions
Time unit 1104
Time values
DBCENTURY setting 1467
DBDATE setting 1469
Time zone
of DBINFO function 1078, 1078
of TODAY operator 1044
Time-limited licenses (IFX_NO_TIMELIMIT_WARNING) 1490
times() operator function 1019
Timezone
setting TZ 1515
TO CLUSTER keywords, in ALTER INDEX statement 84
TO keyword 931
EXTEND function 1426
in ALTER FRAGMENT statement 52
in ALTER INDEX statement 84
in ALTER USER statement 165
in CONNECT statement 176
in CREATE SECURITY POLICY statement 298
in EXTEND function 1101
in GRANT FRAGMENT statement 573
in GRANT statement 558, 560, 563, 567, 568
in INTERVAL Field Qualifier segment 1194
in OUTPUT statement 622
in RENAME COLUMN statement 643
in RENAME DATABASE statement 644, 645
in RENAME INDEX statement 645
in RENAME SECURITY statement 647
in RENAME SEQUENCE statement 648
in RENAME TABLE statement 648
in RENAME TRUSTED CONEXT statement 650
in ROLLBACK WORK statement 684
in SET DEBUG FILE statement 797
in SET EXPLAIN statement 800
in SET ISOLATION statement 812
in SET LOCK MODE statement 817
in SET SESSION AUTHORIZATION statement 831
in UNLOAD statement 863
in WHENEVER statement 907
TO NOT CLUSTER keywords
in ALTER INDEX statement 85
TO_CHAR function 1101
TO_CHAR function, as conversion function 1601
TO_DATE function 1101, 1107
TO_DATE function, as conversion function 1602
TO_NUMBER function 1107
TODAY function
as expression 1040
in ALTER TABLE statement 103
in Condition segment 972
in CREATE TABLE statement 324
in DEFINE statement 922
in INSERT 583, 588
TODAY function, in constant expression 1612, 1664
TODAY operator 1342
Trace class 1376
Trace messages 1377
TRACE statement 959
debugging an SPL routine 1815
output 1833
specifying the output file 797
Trace statements 1377
TRAILING keyword, in TRIM expressions 1121
Transaction isolation level 1388, 1504
TRANSACTION keyword
in CONNECT statement 176
in SET TRANSACTION statement 837
Transaction logging 1334, 1388
contents of log 1693

description of 1692
Transaction mode
constraints 841
Transactions
access mode 840
description of 1691
end of 1751
example 442
example with DELETE 1724
locks held to end of 1741
locks released at end of 1741
logging 374, 1692
partial rollback 684
read-only 840
rollback 683
statements that initiate 833
use signalled in SQLWARN 1705
using cursors in 449
without error handling 169
Transition fragment 26, 31, 38, 56, 67
TRANSITION keyword in ALTER FRAGMENT statement 52, 63
Transition value 56
Transition value of a table fragmented by interval 63
TREE keyword
in ALTER SECURITY LABEL COMPONENT statement 88
in CREATE SECURITY LABEL COMPONENT statement 294
Trigger
inherited 383, 391
overriding 391
Trigger action 378
definition of 1823
REFERENCING clause 1825
Trigger event 378
definition of 1823
DELETE 386
example of 1823
INSERT 386, 396
privileges on 385
SELECT 388
UPDATE 387
Trigger events
recommendation against for auditing 1828
Trigger functions 230
TRIGGER keyword
in DROP TRIGGER statement 491
in EXECUTE FUNCTION statement 502
in EXECUTE PROCEDURE statement 510
Trigger procedures 276
Trigger routines 230, 276, 1360, 1828
Trigger UDR 230, 276
Trigger-type Boolean operator 975
Triggered action
action statements 400
BEFORE and AFTER 1824
cascading 394
correlation names 407
effect of cursors 385
FOR EACH ROW 1825
for multiple triggers 394
generating an error message 1834
in relation to triggering statement 1823
list of actions 399
SELECT statements 1829
statements 1821
tracing 1832
using 1824
using SPL routines 1827
WHEN condition 399, 1826
Triggering statement
consistent results 401
performance 386
UPDATE 387
Triggers
affected by dropping a column from table 135
affected by modifying a column 149
creating 1822
creation-time value 1469, 1471
declaring the name 1823
definition of 1821
disabled 793
enabling or disabling 791
in a table hierarchy 1828
INSTEAD OF 1831
overriding 411
re-entrant 1831
restrictions on Select trigger execution 1830
select
defining on a table hierarchy 1831
Select 1828
sysobjstate data 1357
systrigbody data 1378, 1378
systriggers data 1378
when to use 1822
TRIGGERS keyword, in SET Database Object Mode statement 786, 787, 843
Trigonometric function
ACOS function 1110
ASIN function 1111
ATAN function 1111
ATAN2 function 1112
COS function 1109
SIN function 1109
TAN function 1110
TRIM function 1121
Triple Data Encryption Standard (TDES or DES3) 1087
TRUE setting
BOOLEAN values 1396
sysams table 1327, 1327, 1327, 1327, 1327
TRUNC function 1057
TRUNCATE statement 858, 1659
Truncation, signalled in SQLWARN 1705
TRUSTED CONTEXT keywords
in ALTER TRUSTED CONTEXT statement 161
in CREATE TRUSTED CONTEXT statement 413
in DROP TRUSTED CONTEXT statement 492
in RENAME TRUSTED CONTEXT statement 650
Trusted context object
renaming 650
Trusted contexts
switching user ID 831
Trusted Facility feature 1080
TRUSTED keyword
in CONNECT statement 176, 184
in RENAME TRUSTED CONTEXT statement 650
Trusted-context object
destroying 492
modifying 161
Two-phase commit operations 426
TYPE field
with DESCRIBE INPUT statement 465
with DESCRIBE statement 461
Type hierarchy 290
TYPE keyword 1410
in ALTER TABLE statement 127, 137
in CREATE DISTINCT TYPE statement 199
in CREATE ROW TYPE statement 286
in CREATE TABLE statement 343
in CREATE VIEW statement 421
in CREATE XADATASOURCE TYPE statement 428
in DROP ROW TYPE statement 483
in DROP TYPE statement 493
in DROP XADATASOURCE TYPE statement 496
in GRANT statement 545
in REVOKE statement 661
Typed collection variable 18, 926
Typed table
ADD TYPE clause 127
altering 127
altering serial columns 142, 290
definition of 1581
in FROM clause of SELECT statement 718
inheritance 344
inserting rows 1665
NOT NULL constraint 286
ONLY keyword in DELETE statement 453
selecting from 1582
Typed view 423
TZ environment variable 1515

U

UDT indexes 1512
UID keyword
in ALTER USER statement 165
in CREATE DEFAULT USER statement 198
in CREATE USER statement 416
in GRANT statement 568
Unary arithmetic operators 1439
Unary CONNECT_BY_ROOT operator 746
Unary minus operator (-) 1198
Unary plus operator (+) 1198
Unary PRIOR operator 746
UNBOUNDED FOLLOWING keywords in OLAP window expressions 1189
UNBOUNDED keyword in OLAP window expressions 1189
UNBOUNDED PRECEDING keywords in OLAP window expressions 1189
Unbuffered logging 819
UNCOMMITTED keyword
in SET TRANSACTION statement 838
Uncommitted read 1388
Uncommitted row 813
Uncorrelated subquery 714
UNDEFINED parameter value 1210
UNDER access privilege 286
UNDER keyword
in ALTER SECURITY LABEL COMPONENT statement 88
in CREATE ROW TYPE statement 286
in CREATE SECURITY LABEL COMPONENT statement 294
in CREATE TABLE statement 343
in GRANT statement 545
in REVOKE statement 658, 661
UNDER ON TYPE keywords
in GRANT statement 545
in REVOKE statement 661
Under privilege 540, 546, 658, 661, 1372
Underscore (_)

in SQL identifiers 1231, 1233
in storage object identifiers 1235
Unicode 777
Uninitialized variables 970
UNION keyword, in set operations 1650
UNION operator
in collection subquery 965
in SELECT statement 690, 758
OUT parameter and 1153
restrictions on use 590, 756
UNION operator, display labels with 1653
Union set operation 1650
UNION subquery 756
Union view 424
Unique constraint 1380, 1412, 1412
dropping 130
rules of use 329
Unique index 1350, 1412
Unique keys 1327
UNIQUE keyword
aggregate scope qualifier 1159
in ALTER TABLE statement 106, 117
in CREATE INDEX statement 239
in CREATE TABLE statement 327, 339
in CREATE TEMP TABLE statement 372, 373
in Expression segment 1154, 1166
in OLAP window aggregate expressions 1185
in SAVEPOINT statement 688
in SELECT statement 698
in subquery 981
UNIQUE keyword, in SELECT statement 1542
Unique numeric values
SERIAL data type 1412
SERIAL8 data type 1412
Units of storage size 1146
UNITS operator 1040, 1399, 1424, 1428, 1439
Universal Time (UT) 1078
UNIX
BSD, default print utility 1476
environment variables 1440
PATH environment variable 1507
System V
default print utility 1476
terminfo libraries 1500, 1515
temporary files 1510
TERM environment variable 1514
TERMCAP environment variable 1514
TERMINFO environment variable 1515
UNIX operating system
epochs 1078
mail utility 957
shell script 958
UNKNOWN truth values 984
UNLOAD statement 863, 863, 1466, 1472
UNLOAD TO file 863
Unloading data
from a fixed-text file 220
to a delimited file 219
to an
Informix
file
220
UNLOCK keyword
in CREATE USER statement 416
UNLOCK TABLE statement 867
Unnamed ROW data type
coltype code 1336
declaring 1411
defined 1410
inserting values 1411
Unnamed row data types
field definition 999
literal values 1199
unloading 863, 866
updating fields 882
Unnamed row type, in VALUES clause 1667
unset utility 1443
unsetenv utility 1443
Unsetting an environment variable 1443
untyped 1767
Untyped collection variable 18, 431, 718, 1001, 1215, 1219
Untyped row variable 21
Untyped table 1373
Untyped view 421
Updatable view 426
UPDATABLE_SECONDARY configuration parameter 817
Update clause in MERGE statement 602
Update cursor 1746
locking considerations 440
restricted statements 444
use in DELETE 456
use in UPDATE 881
Update cursor, definition of 1731
UPDATE keyword 574, 1731
in CREATE TRIGGER statement 387
in DECLARE statement 433
in GRANT statement 540
in MERGE statement 602
in REVOKE FRAGMENT statement 681
in REVOKE statement 658
in SELECT statement 690, 771
in SET ISOLATION statement 815
in UPDATE statement 868
in UPDATE STATISTICS statement 883
Update locks 816
Update locks, retaining 1747
Update MERGE operations 1671
Update privilege 540, 658, 1335, 1347, 1372, 1503
Update privilege, with a view 870
UPDATE statement 868, 1380
UPDATE statements 1491
and end of data 1730
and triggers 400
collection data types 1675
collection variables 1219
cursor with 438
description of 1671, 1671
distributed 587, 877
embedded 1731
failures 1691
lock mode 1747
number of rows 1704
OUT parameter and SLVs 1153
preparing 1718
restrictions on subqueries 1673
SET clause 873, 1673
single-column SET clause 873
smart large objects 1677
smart large objects with 1031
update triggers 385
updating through a view 870
WHERE clause 1672
with a supertable 1676
with FETCH 519
with row data types 1674
with SELECT . . . FOR UPDATE 771
with uniform values 1672
UPDATE STATISTICS FOR PROCEDURE statement 1363
UPDATE STATISTICS statement 883, 1352, 1479
and DBUPSPACE environment variable 1479
dropping data distributions 893
effect on sysdistrib table 1344
examining index pages 889
LOW mode 892
specifying distributions only 898
sysindices data 1352
updating system catalog tables 1322
upgrading the database server 902
Update trigger 387, 1378
Updating a specific table in a table hierarchy 870
UPDATING operator 230, 975
Upgrading the database server 887, 902, 902
UPON keyword
in CREATE TRUSTED CONTEXT statement 413
UPPER function 1130
UPPER function, as string manipulation function 1605
Upper index filter 804
Uppercase characters
converting from lowercase 1130
in database server names 178, 1227
Uppercase mode codes 1360
Uppercase privilege codes 1317, 1335, 1372
USAGE keyword
in GRANT statement 545, 546, 549
in REVOKE statement 661, 664
USAGE ON LANGUAGE keywords
in GRANT statement 546, 549
in REVOKE statement 664
USAGE ON TYPE keywords
in GRANT statement 545
in REVOKE statement 661
USE keyword
in ALTER TRUSTED CONTEXT statement 161
in CREATE TRUSTED CONTEXT statement 413
USE_DTENV environment variable 584, 592
USE_HASH keyword, in optimizer directives 1255
USE_NL keyword, in optimizer directives 1255
USELASTCOMMITTED configuration parameter 600
USELASTCOMMITTED configuration parameters 839
USELASTCOMMITTED environment option 817, 819, 839
USELASTCOMMITTED keyword, in SET ENVIRONMENT statement 600
USEOSTIME configuration parameter 1044
User environment variable 1448
USER function
as constant expression 1040
defined 1042
in ALTER TABLE statement 103
in Condition segment 972
in CREATE TABLE statement 324
in DEFINE statement 922
in INSERT statement 583, 583, 588
in Literal Row segment 1199
USER function, in expression 1611, 1664
User informix 199, 471, 555, 1263, 1322, 1332, 1434

as DBSA 668
privileges associated with 539
USER keyword
in CONNECT statement 180
in CREATE DEFAULT USER statement 198
in CREATE TRUSTED CONTEXT statement 413
in CREATE USER statement 416
in DROP USER statement 494
in GRANT statement 558, 560, 563, 567
in RENAME USER statement 653
in REVOKE statement 671, 673, 675, 678
in SET USER PASSWORD statement 844
User name 1388
case-sensitivity 537, 831
mapping to properties 165, 198, 416, 568
unauthorized external users 165, 568
using another name 831
User privileges
syscolauth data 1335
sysfragauth data 1347
syslangauth data 1356
sysprocauth data 1358
systabauth data 1372
sysusers data 1379
sysxtdtypeauth data 1382
User properties 165, 198, 416, 568
User-defined access method
creating 184
modifying 22
User-defined aggregates 1326
creating 185
dropping 470
invoking 1166
User-defined casts 1436
User-defined casts (UDCs) 1332
User-defined data types 995
calculating distribution statistics 890
casting 1436
casting into built-in type 1433
dropping distribution statistics 890
maximum in one row 320, 998
opaque 1433
privileges 545, 545, 662
sysxtddesc data 1382
sysxtdtypes data 1382, 1383
User-defined function 1149
arguments 1209
cursor 505
data types of return value 1272
inserting data with 590
iterator 1277
negator 503, 1277
noncursor 503
OUT parameter 1153
selectivity 1279
USAGE ON LANGUAGE privileges 549, 549, 664, 664
variant 1231, 1280
User-defined routine
USAGE ON LANGUAGE privileges 549, 549, 664, 664
User-defined routines
arguments 1209
casts (syscasts) 1332
check constraints (syscheckudrdep) 1333
defined 273
dropping with DROP ROUTINE 481
error messages (syserrors) 1346
EXTEND role 555, 668
for OPAQUE data types 1408
functional index 1480
ill-behaved 1276
in SELECT statements 707
inserting data with 590
language authorization (syslangauth) 1356
ownership of created objects 235, 1229
privileges 546, 662, 1358, 1503
protected 1360
REFERENCES keyword with BYTE or TEXT data type 929
reoptimization 899
RESTRICTED mode 831
return values 1271
secondary access method 1341
sysprocedures data 1360
User-defined VP class 1276
USERMAPPING configuration parameter 165, 198, 568
USETABLENAME environment variable 95, 489, 489, 859, 1516
USING BSON keywords
in CREATE INDEX statement 243
USING DESCRIPTOR keywords
in FETCH 518
in OPEN 616
in PUT 502
USING keyword 1688
in CONNECT statement 180
in CREATE EXTERNAL TABLE statement 201
in CREATE INDEX statement 237
in CREATE TABLE statement 367
in CREATE TRUSTED CONTEXT statement 413
in CREATE XADATASOURCE statement 426
in DELETE statement 451
in DESCRIBE INPUT statement 465
in DESCRIBE statement 459, 461
in EXECUTE statement 498, 501
in FETCH statement 513
in MERGE statement 602
in OPEN statement 616, 619
in PUT statement 634
in START VIOLATIONS TABLE statement 844
USING SQL DESCRIPTOR keywords
in DESCRIBE INPUT statement 463, 465
in DESCRIBE statement 459, 461
in EXECUTE statement 501
Using the GROUP BY and HAVING Clauses 1620
USTLOW_SAMPLE configuration parameter 892
USTLOW_SAMPLE session environment variable 892
UT (Universal Time) 1078
UTC (Coordinated Universal Time) 1078
UTC time and time zone, returned by DBINFO function 1613
UTC_CURRENT option of DBINFO 1078
UTC_TO_DATETIME option of DBINFO 1078
UTF-8 character encoding 777, 777
UTF-8 locale 1092
Utilities
cdr 98, 98, 728
chkenv 1442, 1444
DB-Access 1322, 1385, 1445, 1466, 1472, 1498
dbexport 817
dbload 1395, 1396
dbschema 94, 1360, 1360
env 1444
export 1443
gcc 1493
getenv 1441
ifx_getenv 1445
ifx_putenv 1445
lp 1476
lpr 1476
oncheck 264, 1090
ondblog 819
oninit 1484
onload 1395, 1396
onmode 833
onsecurity 1489
onspaces 23, 251, 575, 680
onstat 796, 833
onutil 1090
printenv 1444
putenv 1441
set 1447
setenv 179, 1443
setnet32 178
Setnet32 1445
source 1442
unset 1443
unsetenv 1443, 1481
vi 1472
Utilities dbload 1415
Utility program
onload 1693
onunload 1693

V

VALUES clause
effect with PUT 636
in INSERT statement 583, 1663
in MERGE statement 602, 1664
NULL values 1667
restrictions 1664
selected columns 1665
valid values 1664
VALUES keyword
in ALTER FRAGMENT statement 27, 27, 42, 47, 52
in CREATE INDEX statement 361
in CREATE TABLE statement 361
in MERGE statement 602
VALUES keyword, in INSERT statement 583
VARCHAR data type 986
([ ]), brackets
MATCHES range delimiters 1417
CHAR data type
collation 1417
Code sets
collation order 1417
East Asian 1417
Collation
VARCHAR data type 1417
coltype code 1336
defined 1417
in LOAD statement 593
Locales
collation order 1417
MATCHES operator 1417
Multibyte characters
VARCHAR data type 1417
nonprintable characters 1417
SQL_LOGICAL_CHAR configuration parameter 1417
storing numeric values 1417
VARCHAR data type

collation 1417
multibyte characters 1417
Zero (0)
C null as terminator 1417
VARCHAR data type, using LENGTH function on 1610
VARIABLE keyword, in CREATE OPAQUE TYPE statement 263
Variable-length opaque data types 1336
Variable-length packets 1491
Variable-length UDT 320, 998, 1383
Variables
BLOB keyword
in DEFINE statement 920
BYTE keyword
in DEFINE statement 920
CLOB keyword
in DEFINE statement 920
COLLECTION keyword
in DEFINE statement 920
declaring in SPL 920
DEFAULT keyword
in DEFINE statement 920
default values in SPL 922, 924
defining and using in SPL routine 1765
global 921
GLOBAL keyword, in DEFINE statement 920
LIKE keyword
in DEFINE statement 920
local 920, 924
NULL keyword
in DEFINE statement 920
PROCEDURE keyword
DEFINE statement 920
PROCEDURE type 929
REFERENCES keyword
in DEFINE statement 920
scope in SPL routine 1765
TEXT keyword
in DEFINE statement 920
uninitialized 961, 970
unknown values in IF 942
with same name as a keyword 1770
VARIANCE function 1154, 1164
VARIANCE function, as aggregate function 1595
VARIANCE window function 1185
Variant function 112, 1231, 1280
Variant function as DISTINCT or OPAQUE value
in ALTER TABLE statement 103
VARIANT keyword
External Routine Reference segment 1229
VARIANT routine 1360
variant SPL function 1759
Varying-length opaque data type 264
VERCOLS keyword
in CREATE TABLE statement 310, 346, 348
Version number, returned by DBINFO function 1613
Version of a table 1373
vi text editor 1472
View
affected by dropping a column 136
affected by modifying a column 149
columns view 1386
creating a synonym for 305
creating a view 421
definition of 1528
deleting in a 1831
dependent 494
dropped by ALTER FRAGMENT statement 35
dropping 494
dropping a synonym 488
Information Schema 1384
inserting into a 1831
INSTEAD OF trigger on a 1832
materialized 421
privileges 544
server_info view 1388
sql_languages view 1387
sysdepend data 1343
sysindexes view 1352
syssynonyms data 1370
syssyntable data 1371
systabauth data 1372
systables data 1373
sysviews data 1380
tables view 1386
typed 286, 423
union 424
untyped 421
updatable 426
updating 870
updating in a 1831
VIEW keyword
in CREATE VIEW statement 421
in DROP VIEW statement 494
Views
restrictions on view definitions 423
Violation detection 1684
Violations
sysobjstate data 1357
sysviolations data 1380
VIOLATIONS keyword
in START VIOLATIONS TABLE statement 844
in STOP VIOLATIONS TABLE statement 856
Violations table 1688
assigning a name 1688
creating 844
declaring a name 846
default name 846
description of 1688
effect on transactions 846
example of privileges 1689
examples 851, 857, 1685
examples of starting 1688
how to stop 856
privileges 849
relationship to diagnostics table 848
relationship to target table 848
restriction on dropping 491
schema 847
security label protection 847
starting 1688
Virtual column 424
Virtual index 184
Virtual machine 1461, 1502
Virtual processors 1512
Virtual table 1212
Virtual-processor class 1276
VISIBLE keyword
in SET INDEXES statement 262
in VISIBLE AND INVISIBLE options for indexes 262

W

WAIT keyword, in SET LOCK MODE statement 817
WARNING keyword
in SET ISOLATION statement 813
Warning message 1346, 1466
Warnings, with SPL routine at compile time 1814
Weekday argument to ADD_MONTHS function 1104
WEEKDAY function 1101
as time function 1596, 1598
using 1598
Well-behaved C UDRs 1276
WHEN keyword
in CASE statement 916
in CREATE TRIGGER statement 400
in EXIT statement 929
in Expression segment 1034, 1035
in MERGE statement 602
WHENEVER statement
syntax and use 904
WHERE 1415
WHERE clause
Boolean expression in 1553
comparison condition 1547
date-oriented functions in 1598
description of 1547
equal sign relational operator 1548
estimated selectivity 895
host variables in 1707
in DELETE 1659
in SELECT statement 690
in system-descriptor area 19
in UPDATE statement 1672
joining tables 736
less-than relational operator 1549
not-equal relational operator 1549
relational operators 1547
selecting a range of characters 1558
subqueries in 1639
wildcard comparisons 1554
with a subquery 732
with ALL keyword 736
with ANY keyword 736
with BETWEEN keyword 733
with IN keyword 733
with IS keyword 734
with NOT keyword 1550
with OR keyword 1551
with relational operator 732
with SOME keyword 736
with string literals 732
WHERE CURRENT OF clause
in DELETE statement 1725
in UPDATE statement 1731
WHERE CURRENT OF keywords
in DELETE statement 451
in UPDATE statement 868, 881
optimizer directives 1248
WHERE keyword 1322, 1333
in DELETE statement 451
in UPDATE statement 868, 868
null data tests 1552
range of values 1549
WHILE keyword
in CONTINUE statement 919
in EXIT statement 929
WHILE statement 961
White space characters
in delimited identifiers 1233
SQL statements 3
Whitespace in identifiers 1481
Wildcard character
brackets ([ ]) 977

asterisk 1538
asterisk (*) 977
backslash (\) 976, 977
caret (^) 977
percent sign (%) 976
protecting 1558
question mark (?) 977
with LIKE 734, 976
with LIKE or MATCHES 1205
with MATCHES 734, 977
Wildcard comparison in WHERE clause 1554
Wildcard, using single character 1555
Window borders 1500
Window Frame clause 1189
Window ORDER clause 1189
Window PARTITION clause 1189
Windows
batch file 958
sqlhosts subkey 178
system commands 958
Windows environments
manipulating environment variables 1445
setting environment variables 1445
WITH APPEND keywords
in SET DEBUG FILE statement 797
WITH AUDIT keywords
in CREATE TABLE statement 346
WITH BUFFERED LOG keywords
in CREATE DATABASE statement 190
WITH CHECK OPTION keywords, in CREATE VIEW statement 421, 425
WITH clause, in SPL function 1759
WITH CONCURRENT TRANSACTION keywords
in CONNECT statement 182
WITH CRCOLS keywords
in CREATE TABLE statement 346
WITH ERKEY keywords
in CREATE TABLE statement 347
WITH ERROR keyword
in SET INDEXES statement 810
WITH ERROR keywords
in ALTER TABLE statement 108
in CREATE INDEX statement 258
in CREATE TABLE statement 336, 337
in SET Database Object Mode statement 784, 788
with FOREACH 935
WITH GRANT OPTION keywords
in GRANT FRAGMENT statement 573
in GRANT statement 537
WITH HOLD keywords 442, 935
in DECLARE statement 433, 449
WITH HOLD keywords, declaring a hold cursor 1751
WITH IDSLBACRULES keywords
in CREATE SECURITY POLICY statement 298
WITH keyword
in ALLOCATE DESCRIPTOR statement 19, 20
in ALTER FRAGMENT statement 41
in ALTER FUNCTION statement 82
in ALTER PROCEDURE statement 85
in ALTER ROUTINE statement 87
in ALTER SEQUENCE statement 94
in ALTER TABLE statement 113, 144
in ALTER TRUSTED CONTEXT statement 161
in ALTER USER statement 165
in CONNECT statement 176
in CREATE AGGREGATE statement 185
in CREATE CAST statement 188
in CREATE DATABASE statement 190
in CREATE DEFAULT USER statement 198
in CREATE FUNCTION statement 225, 225
in CREATE INDEX statement 249, 258
in CREATE PROCEDURE statement 270
in CREATE SECURITY POLICY statement 298
in CREATE SEQUENCE statement 304
in CREATE TABLE statement 310, 320, 322, 346, 346, 346, 347, 347, 348
in CREATE TRUSTED CONTEXT statement 413
in CREATE USER statement 416
in CREATE VIEW statement 421, 425
in DECLARE statement 433, 449
in EXECUTE FUNCTION statement 502
in EXECUTE PROCEDURE statement 510
in GRANT FRAGMENT statement 573
in GRANT statement 537
in OPEN statement 616
in SELECT statement 743
in SET CONSTRAINTS statement 782
in SET Database Object Mode statement 784, 788
in SET DEBUG FILE statement 797
in SET ISOLATION statement 813
WITH LISTING IN clause, use in SPL routine 1761
WITH LISTING IN keywords
in CREATE FUNCTION statement 225
in CREATE PROCEDURE statement 270
WITH LOG keyword
in CREATE DATABASE statement 190
WITH LOG MODE ANSI keywords
in CREATE DATABASE statement 190
WITH MAX keywords
in ALLOCATE DESCRIPTOR statement 19, 20
WITH NO LOG keywords
in CREATE TEMP TABLE statement 370
WITH REOPTIMIZATION keywords in OPEN statement 616
WITH REPLCHECK keywords
in CREATE TABLE statement 347
WITH RESUME keywords
in ON EXCEPTION statement 949
in RETURN statement 954
WITH ROWIDS keywords
in ALTER FRAGMENT statement 41
in CREATE TABLE statement 356
WITH TRIGGER REFERENCES keywords
in EXECUTE FUNCTION statement 502
in EXECUTE PROCEDURE statement 510
WITH USE FOR keywords
in CREATE TRUSTED CONTEXT statement 413
WITH VERCOLS keywords
in CREATE TABLE statement 348
WITH WARNING keywords
in SET ISOLATION statement 813
WITHOUT AUTHENTICATION keywords
in ALTER TRUSTED CONTEXT statement 161
in CREATE TRUSTED CONTEXT statement 413
WITHOUT ERROR keywords
in ALTER TABLE statement 108
in CREATE INDEX statement 258
in CREATE TABLE statement 336, 337
in SET Database Object Mode statement 784, 788
ON DELETE CASCADE keywords
in ALTER TABLE statement 108
WITHOUT ERROR keywords keyword
in SET INDEXES statement 810
WITHOUT HEADINGS keywords
in OUTPUT statement 622
WITHOUT keyword
in ALTER TABLE statement 108
in ALTER TRUSTED CONTEXT statement 161
in BEGIN WORK statement 168
in CREATE INDEX statement 258
in CREATE TABLE statement 336
in CREATE TRUSTED CONTEXT statement 413
in OUTPUT statement 622
in SET CONSTRAINTS statement 782
in SET Database Object Mode statement 784, 788
WORK keyword
in BEGIN WORK statement 168
in COMMIT WORK statement 175
in ROLLBACK WORK statement 684
WORK WITHOUT REPLICATION keywords, in BEGIN WORK statement 168
WRITE keyword
in CREATE SECURITY POLICY statement 298
in GRANT statement 563
in REVOKE statement 675
in SET TRANSACTION statement 837
Write lock 440
Write-access rules for label-based access 298
WRITEDOWN keyword
in GRANT statement 560
in REVOKE statement 673
WRITEUP keyword
in GRANT statement 560
in REVOKE statement 673
Writing direction 1143

X

X for storage in an extent space 184, 1267
X setting
sysams.am_sptype 1327
systabauth.tabauth 1372
X/Open
compliance 1388
server_info view 1388
X/Open CAE standards 1384
X/Open DTP XA standard 426, 426, 428
X/Open mode
CONNECT statement 181
FETCH statement 513
OPEN statement 620
XA data source
privileges to create 426
privileges to drop 495
XA data source type
creating 428
dropping 496
XA data source types 1381
XA data sources 1381
XA Switch Structure 1270
xa.h file 1270
XADATASOURCE keyword
in CREATE XADATASOURCE statement 426
in CREATE XADATASOURCE TYPE statement 428

in DROP XADATASOURCE statement 495
in DROP XADATASOURCE TYPE statement 496
XID data type 700, 986
XML (Extensible Markup Language) 1398
XML messages 911
XPG4 standard 1386, 1386

Y

Y
value of MORE field 530
Y setting
DBDATE 1469
sysroleauth.is_grantable 1365
Year 2000 1467
YEAR function 1104
as time function 1596
using 1599
YEAR keyword 1194, 1194, 1194
EXTEND function 1426
Year values, two and four digit 1467, 1469
YES
NODEFDAC setting 544
yes setting
NODEFDAC 1503
YES setting
columns.is_nullable 1386
sql_languages.integrity 1387

Z

Zero
DIRECTIVES setting 1246
EXT_DIRECTIVES setting 686
extent size encoding 1352
fextsize or nextsize default value 252
IFX_EXTDIRECTIVES setting 686
IFX_EXTEND_ROLE setting 668
in UNLOAD file 863
invalid divisor 527
prohibited MOD divisor 1060
scale and ROUND function 1061
scale and TRUNC function 1066
setting of STMT_CACHE_NOLIMIT 836
sqlca value after INSERT 590
sqlcode value after ALLOCATE COLLECTION 18
subseconds and CURRENT function 1044
Sunday and WEEKDAY function 1104
sysdirectives.active value 688
time unit value returned by EXTEND 1106
to specify next serial value 585
variance and STDEV function 1163
variance and VARIANCE function 1164
Zero (0)
DBDATE separator 1469
DECIMAL scale 1399
hexadecimal digit 1472
IFX_DIRECTIVES setting 1485, 1485
IFX_LARGE_PAGES setting 1486
IFX_LONGID setting 1488
IFX_NETBUF_PVTPOOL_SIZE setting 1488
integer scale 1386, 1399
OPTCOMPIND setting 1504
OPTMSG setting 1505
padding of 1-digit years 1467
padding with DBFLTMASK 1472
PDQPRIORITY setting 1507
PSORT_NPROCS setting 1512
STMT_CACHE setting 1513
sysams values 1327, 1327, 1327, 1327, 1327
sysfragments.hybdpos 1348
sysindices.nrows 1352
systables.type_xid 1373
sysxdtypes values 1383