

# **HCL Informix 14.10 - SQL programming Guide**



# Contents

<b>Chapter 1. SQL programming.....</b>	<b>3</b>
SQL programming.....	
SQL programming.....	
SQL programming.....	
SQL programming.....	
SQL programming.....	
SQL programming.....	
SQL programming.....	
SQL programming.....	
SQL programming.....	
Guide to SQL: Reference.....	3
System catalog tables.....	3
Data types.....	81
Environment variables.....	140
Appendixes.....	215
Guide to SQL: Tutorial.....	220
Database concepts.....	220
Compose SELECT statements.....	232
Select data from complex types.....	280
Functions in SELECT statements.....	292
Compose advanced SELECT statements.....	320
Modify data.....	358
Access and modify data in an external database.....	396
SQL programming.....	400
Modify data through SQL programs.....	423
Programming for a multiuser environment.....	433
Create and use SPL routines.....	453
Create and use triggers.....	522
<b>Index.....</b>	<b>538</b>

# 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: Reference

These topics contain 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.

These topics are taken from the *HCL® Informix® Guide to SQL: Reference*.

## 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**  $\geq$  100. System catalog tables all have a **tabid**  $<$  100. See later in this section and [SYSTABLES on page 65](#) 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
<b>tablename</b>	customer	california
<b>owner</b>	maryl	maryl
<b>partnum</b>	16778361	0
<b>tabid</b>	101	102
<b>rowsize</b>	134	134
<b>ncols</b>	10	4
<b>nindexes</b>	2	0
<b>nrows</b>	0	0
<b>created</b>	01/26/2007	01/26/2007
<b>version</b>	1	0
<b>tabtype</b>	T	V

Column name	First row	Second row
<b>locklevel</b>	P	B
<b>npused</b>	0	0
<b>fectsize</b>	16	0
<b>nextsize</b>	16	0
<b>flags</b>	0	0
<b>site</b>		
<b>dbname</b>		

Each table recorded in the **sysables** 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		

idxname	c_num_ix	state_ix
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 DB-Access, 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 DataBlade® 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 13](#)

---

[SYSAMS on page 14](#)

---

[SYSATTRTYPES on page 17](#)

---

[SYSAUTOLOCATE on page 18](#)

---

[SYSBLOBS on page 19](#)

---

[SYSCASTS on page 20](#)

---

[SYSCHECKS on page 20](#)

---

[SYSCHECKUDRDEP on page 21](#)

---

[SYSCOLATTRIBS on page 21](#)

---

[SYSCOLAUTH on page 22](#)

---

[SYSCOLDEPEND on page 23](#)

---

[SYSCOLUMNS on page 23](#)

---

[SYSCONSTRAINTS on page 28](#)

---

[SYSDEFAULTS on page 29](#)

---

[SYSDEPEND on page 30](#)

---

[SYSDIRECTIVES on page 31](#)

---

[SYSDISTRIB on page 31](#)

---

---

**System Catalog Tables**

---

[SYSDOMAINS on page 33](#)

---

[SYSERRORS on page 34](#)

---

[SYSEXTCOLS on page 34](#)

---

[SYSEXTDFILES on page 35](#)

---

[SYSEXTTERNAL on page 35](#)

---

[SYSFRAGAETH on page 36](#)

---

[SYSFRAGDIST on page 37](#)

---

[SYSFRAGMENTS on page 39](#)

---

[SYSINDEXES on page 41](#)

---

[SYSINDICES on page 43](#)

---

[SYSINHERITS on page 46](#)

---

[SYSLANGAUTH on page 46](#)

---

[SYSLOGMAP on page 47](#)

---

[SYSOBJSTATE on page 47](#)

---

[SYSOPCLASSES on page 48](#)

---

[SYSOPCLSTR on page 48](#)

---

[SYSPROCAUTH on page 50](#)

---

[SYSPROCBODY on page 51](#)

---

[SYSPROCCOLUMNS on page 52](#)

---

[SYSPROCEDURES on page 52](#)

---

[SYSPROCPLAN on page 55](#)

---

[SYSREFERENCES on page 56](#)

---

[SYSROLEAUTH on page 56](#)

---

[SYSROUTINELANGS on page 57](#)

---

[SYSSECLABELAUTH on page 57](#)

---

[SYSSECLABELCOMPONENTS on page 58](#)

---

[SYSSECLABELCOMPONENTELEMENTS on page 58](#)

---

[SYSSECLABELNAMES on page 59](#)

---

---

**System Catalog Tables**

---

[SYSSECLABELS on page 59](#)

---

[SYSSECPOLICIES on page 59](#)

---

[SYSSECPOLICYCOMPONENTS on page 60](#)

---

[SYSSECPOLICYEXEMPTIONS on page 60](#)

---

[SYSSEQUENCES on page 61](#)

---

[SYSSURROGATEAUTH on page 61](#)

---

[SYSSYNONYMS on page 62](#)

---

[SYSSYNTABLE on page 63](#)

---

[SYSTABAMDATA on page 63](#)

---

[SYSTABAUTH on page 64](#)

---

[SYSTABLES on page 65](#)

---

[SYSTRACECLASSES on page 68](#)

---

[SYSTRACEMSGS on page 69](#)

---

[SYSTRIGBODY on page 69](#)

---

[SYSTRIGGERS on page 70](#)

---

[SYSUSERS on page 71](#)

---

[SYSVIEWS on page 72](#)

---

[SYSVIOLATIONS on page 72](#)

---

[SYSXADATASOURCES on page 73](#)

---

[SYSXASOURCETYPES on page 73](#)

---

[SYSXTDDESC on page 74](#)

---

[SYSXTDTYPEAUTH on page 74](#)

---

[SYSXTDTYPES on page 75](#)

---

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 the [Data types on page 81](#) 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:

<i>Table_name.Column_name</i>	<b>Data type</b>
<b>sysams.am_sptype</b>	CHAR(3)
<b>syscolauth.colauth</b>	CHAR(3)
<b>sysdefaults.class</b>	CHAR(1)
<b>sysfragauth.fragauth</b>	CHAR(6)
<b>sysinherits.class</b>	CHAR(1)
<b>syslangauth.langauth</b>	CHAR(1)
<b>sysprocauth.procauth</b>	CHAR(1)
<b>sysprocedures.mode</b>	CHAR(1)
<b>systabauth.tabauth</b>	CHAR(9)
<b>systriggers.event</b>	CHAR(1)
<b>sysxdttypeauth.auth</b>	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 1. SYSAGGREGATES table column descriptions**

<b>Column</b>	<b>Type</b>	<b>Explanation</b>
<b>name</b>	VARCHAR(128)	Name of the aggregate
<b>owner</b>	CHAR(32)	Name of the owner of the aggregate

**Table 1. SYSAGGREGATES table column descriptions**

(continued)

Column	Type	Explanation
<b>aggid</b>	SERIAL	Unique code identifying the aggregate
<b>init_func</b>	VARCHAR(128)	Name of initialization UDR
<b>iter_func</b>	VARCHAR(128)	Name of iterator UDR
<b>combine_func</b>	VARCHAR(128)	Name of combine UDR
<b>final_func</b>	VARCHAR(128)	Name of finalization UDR
<b>handlesnulls</b>	BOOLEAN	NULL-handling indicator: <ul style="list-style-type: none"> <li>• t = handles NULLs</li> <li>• f = does not handle NULLs</li> </ul>

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 2. SYSAMS table column descriptions**

Column	Type	Explanation
<b>am_name</b>	VARCHAR(128, 0)	Name of the access method
<b>am_owner</b>	CHAR(32)	Name of the owner of the access method
<b>am_id</b>	INTEGER	Unique identifying code for an access method  This corresponds to the <b>am_id</b> columns in the <b>systables</b> , <b>sysindices</b> , and <b>sysopclasses</b> tables.

Table 2. SYSAMS table column descriptions (continued)

Column	Type	Explanation
<b>am_type</b>	CHAR(1)	Type of access method: P = Primary; S = Secondary
<b>am_sptype</b>	CHAR(3)	Types of spaces where the access method can exist: <ul style="list-style-type: none"> <li>• <b>A</b> means the access method supports extspaces and sbspaces. If the access method is built in, such as a B-tree, it also supports dbspaces.</li> <li>• <b>D</b> or <b>d</b> means the access method supports dbspaces only.</li> <li>• <b>DS</b> means the access method supports dbspaces and sbspaces.</li> <li>• <b>S</b> or <b>s</b> means the access method supports sbspaces only.</li> <li>• <b>X</b> or <b>x</b> means the access method supports extspaces only.</li> <li>• <b>SX</b> means the access method supports sbspaces and extspaces.</li> </ul>
<b>am_defopclass</b>	INTEGER	Unique identifying code for default-operator class  Value is the <b>opclassid</b> from the entry for this operator class in the <b>sysopclasses</b> table.
<b>am_keyscan</b>	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 <b>am_getnext</b> function.) (0 = FALSE; Non-zero = TRUE)
<b>am_unique</b>	INTEGER	Whether a secondary access method can support unique keys (0 = FALSE; Non-zero = TRUE)
<b>am_cluster</b>	INTEGER	Whether a primary access method supports clustering (0 = FALSE; Non-zero = TRUE)
<b>am_rowids</b>	INTEGER	Whether a primary access method supports rowids (0 = FALSE; Non-zero = TRUE)
<b>am_readwrite</b>	INTEGER	Whether a primary access method can both read and write ( 0 = access method is read-only; Non-zero = access method is read/write )
<b>am_parallel</b>	INTEGER	Whether an access method supports parallel execution (0 = FALSE; Non-zero = TRUE)
<b>am_costfactor</b>	SMALLFLOAT	The value to be multiplied by the cost of a scan to normalize it to costing done for built-in access methods

Table 2. SYSAMS table column descriptions (continued)

Column	Type	Explanation
		The scan cost is the output of the <b>am_scancost</b> function.
<b>am_create</b>	INTEGER	The routine specified for the AM_CREATE purpose for this access method  Value = <b>procid</b> for the routine in the <b>sysprocedures</b> table.
<b>am_drop</b>	INTEGER	The routine specified for the AM_DROP purpose function for this access method
<b>am_open</b>	INTEGER	The routine specified for the AM_OPEN purpose function for this access method
<b>am_close</b>	INTEGER	The routine specified for the AM_CLOSE purpose function for this access method
<b>am_insert</b>	INTEGER	The routine specified for the AM_INSERT purpose function for this access method
<b>am_delete</b>	INTEGER	The routine specified for the AM_DELETE purpose function for this access method
<b>am_update</b>	INTEGER	The routine specified for the AM_UPDATE purpose function for this access method
<b>am_stats</b>	INTEGER	The routine specified for the AM_STATS purpose function for this access method
<b>am_scancost</b>	INTEGER	The routine specified for the AM_SCANCOST purpose function for this access method
<b>am_check</b>	INTEGER	The routine specified for the AM_CHECK purpose function for this access method
<b>am_beginscan</b>	INTEGER	The routine specified for the AM_BEGINSCAN purpose function for this access method
<b>am_endscan</b>	INTEGER	The routine specified for the AM_ENDSCAN purpose function for this access method
<b>am_rescan</b>	INTEGER	The routine specified for the AM_RESCAN purpose function for this access method
<b>am_getnext</b>	INTEGER	The routine specified for the AM_GETNEXT purpose function for this access method

**Table 2. SYSAMS table column descriptions (continued)**

Column	Type	Explanation
<b>am_getbyid</b>	INTEGER	The routine specified for the AM_GETBYID purpose function for this access method
<b>am_build</b>	INTEGER	The routine specified for the AM_BUILD purpose function for this access method
<b>am_init</b>	INTEGER	The routine specified for the AM_INIT purpose function for this access method
<b>am_truncate</b>	INTEGER	The routine specified for the AM_TRUNCATE purpose function for this access method
<b>am_expr_pushdown</b>	INTEGER	Reserved for future use 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 3. SYSATTRTYPES table column descriptions**

Column	Type	Explanation
<b>extended_id</b>	INTEGER	Identifying code of an extended data type  Value is the same as in the <b>sysxdtypes</b> table ( <a href="#">SYSXTDTYPES on page 75</a> ).
<b>seqno</b>	SMALLINT	Identifying code of an entry having <b>extended_id</b> type
<b>levelno</b>	SMALLINT	Position of member in collection hierarchy
<b>parent_no</b>	SMALLINT	Value in the <b>seqno</b> column of the complex data type that contains this member
<b>fieldname</b>	VARCHAR(128)	Name of the field in a row type

**Table 3. SYSATTRTYPES table column descriptions (continued)**

Column	Type	Explanation
		Null for other complex data types
<b>fieldno</b>	SMALLINT	Field number sequentially assigned by system (from left to right within each row type)
<b>type</b>	SMALLINT	Code for the data type  See the description of <b>syscolumns.coltype</b> (page <a href="#">SYSCOLUMNS on page 23</a> ).
<b>length</b>	SMALLINT	Length (in bytes) of the member
<b>xtd_type_id</b>	INTEGER	Code identifying this data type  See the description of <b>sysxdtypes.extended_id</b> ( <a href="#">SYSXTDTYPES on page 75</a> ).

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. The **sysautolocate** system catalog table is reserved for future use.

**Table 4. SYSAUTOLOCATE table column descriptions**

Column	Type	Explanation
<b>dbnum</b>	INTEGER	The ID number of the dbspace. 0 indicates multiple dbspaces. Reserved for future use.
<b>dbname</b>	VARCHAR(128,0)	The name of the dbspace. An asterisk (*) indicates multiple dbspaces. Reserved for future use.
<b>pagesize</b>	SMALLINT	The page size of the dbspace. 0 indicates multiple page sizes. Reserved for future use.
<b>flags</b>	INTEGER	<ul style="list-style-type: none"> <li>• 1 = On. The dbspace is available for automatic table fragmentation.</li> <li>• 2 = Off. The dbspace is not available for automatic table fragmentation.</li> </ul>

Reserved for future use.

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 5. SYSBLOBS table column descriptions**

Column	Type	Explanation
<b>spacename</b>	VARCHAR(128)	Name of partition, dbspace, or family
<b>type</b>	CHAR(1)	Code identifying the type of storage media: M = Magnetic Code identifying the type of storage media: M = Magnetic O = Optical
<b>tabid</b>	INTEGER	Code identifying the table
<b>colno</b>	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 6. SYSCASTS table column descriptions**

Column	Type	Explanation
<b>owner</b>	CHAR(32)	Owner of cast (user <b>informix</b> for built-in casts and <i>user</i> name for implicit and explicit casts)
<b>argument_type</b>	SMALLINT	Source data type on which the cast operates
<b>argument_xid</b>	INTEGER	Code for the source data type specified in the <b>argument_type</b> column
<b>result_type</b>	SMALLINT	Code for the data type returned by the cast
<b>result_xid</b>	INTEGER	Data type code of the data type named in the <b>result_type</b> column
<b>routine_name</b>	VARCHAR(128)	Function or procedure implementing the cast
<b>routine_owner</b>	CHAR(32)	Name of owner of the function or procedure specified in the <b>routine_name</b> column
<b>class</b>	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 7. SYSCHECKS table column descriptions**

Column	Type	Explanation
<b>constrid</b>	INTEGER	Unique code identifying the constraint
<b>type</b>	CHAR(1)	Form in which the check constraint is stored: B = Binary encoded s = Select T = Text
<b>seqno</b>	SMALLINT	Line number of the check constraint



**Table 7. SYSCHECKS table column descriptions (continued)**

Column	Type	Explanation
<b>checktext</b>	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 8. SYSCHECKUDRDEP table column descriptions**

Column	Type	Explanation
<b>udr_id</b>	INTEGER	Unique code identifying the UDR
<b>constraint_id</b>	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 **syscolattrs** 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 9. SYSCOLATTRIBS table column descriptions**

Column	Type	Explanation
<b>tabid</b>	INTEGER	Code uniquely identifying the table
<b>colno</b>	SMALLINT	Number of the column that contains the smart large object
<b>extentsize</b>	INTEGER	Pages in smart-large-object extent, expressed in KB

**Table 9. SYSCOLATTRIBS table column descriptions**

(continued)

Column	Type	Explanation
<b>flags</b>	INTEGER	Integer representation of the combination (by addition) of hexadecimal values of the following parameters: <ul style="list-style-type: none"> <li>• LO_NOLOG (0x00000001 = 1) = The smart large object is not logged.</li> <li>• LO_LOG (0x00000010 = 2) = Logging of smart large objects conforms to current log mode of the database.</li> <li>• LO_KEEP_LASTACCESS_TIME (0x00000100 = 4) = Keeps a record of when this column was most recently accessed by a user.</li> <li>• LO_NOKEEP_LASTACCESS_TIME (0x00001000 = 8) = No record is kept of when this column was most recently accessed by a user.</li> <li>• HI_INTEG (0x00010000= 16) = Sbspace data pages have headers and footers to detect incomplete writes and data corruption.</li> <li>• MODERATE_INTEG (0x00100000= 32) = Data pages have headers but no footers.</li> </ul>
<b>flags1</b>	INTEGER	Reserved for future use
<b>sbspace</b>	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
<b>grantor</b>	VARCHAR(32)	Authorization identifier of the grantor
<b>grantee</b>	VARCHAR(32)	Authorization identifier of the grantee
<b>tabid</b>	INTEGER	Code uniquely identifying the table
<b>colno</b>	SMALLINT	Column number within the table
<b>colauth</b>	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
<b>constrid</b>	INTEGER	Code uniquely identifying the constraint
<b>tabid</b>	INTEGER	Code uniquely identifying the table
<b>colno</b>	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 21](#), which lists every check constraint that is referenced by a user-defined routine.

See also the **sysreferences** table in [SYSREFERENCES on page 56](#), which describes dependencies of referential constraints.

## SYSCOLUMNNS

The **syscolumnns** system catalog table describes each column in the database.

One row exists for each column that is defined in a table or view.

**Table 10. The SYSCOLUMNNS table**

Column	Type	Explanation
<b>colname</b>	VARCHAR(128)	Column name
<b>tabid</b>	INTEGER	Identifying code of table containing the column
<b>colno</b>	SMALLINT	Column number
		The system sequentially assigns this (from left to right within each table).

**Table 10. The SYSCOLUMNS table (continued)**

Column	Type	Explanation
<b>coltype</b>	SMALLINT	Code indicating the data type of the column:
		0 = CHAR
		1 = SMALLINT
		2 = INTEGER
		3 = FLOAT
		4 = SMALLFLOAT
		5 = DECIMAL
		6 = SERIAL <sup>1</sup>
		7 = DATE
		8 = MONEY
		9 = NULL
		10 = DATETIME
		11 = BYTE
		12 = TEXT
		13 = VARCHAR
		14 = INTERVAL
		15 = NCHAR
		16 = NVARCHAR
		17 = INT8
		18 = SERIAL8 <sup>1</sup>
		19 = SET
		20 = MULTISSET
		21 = LIST
		22 = ROW (unnamed)
		23 = COLLECTION
		40 = LVARCHAR fixed-length opaque types <sup>2</sup>
		41 = BLOB, BOOLEAN, CLOB variable-length opaque types <sup>2</sup>
		43 = LVARCHAR (client-side only)
		45 = BOOLEAN
		52 = BIGINT
		53 = BIGSERIAL <sup>1</sup>
		2061 = IDSSECURITYLABEL <sup>2,3</sup>
		4118 = ROW (named)

Table 10. The SYSCOLUMNS table (continued)

Column	Type	Explanation
<b>collength</b>	Any of the following data types: <ul style="list-style-type: none"> <li>• Integer-based</li> <li>• Varying-length character</li> <li>• Time</li> <li>• Fixed-point</li> <li>• Simple-large-object</li> <li>• <a href="#">IDSSECURITYLABEL on page 100</a></li> </ul>	The value depends on the data type of the column. For some data types, the value is the column length (in bytes). See <a href="#">Storing Column Length on page 27</a> for more information.
<b>colmin</b>	INTEGER	Holds the second-smallest value in a column
<b>colmax</b>	INTEGER	Holds the second-largest value in a column
<b>extended_id</b>	INTEGER	Data type code, from the <b>sysxdtypes</b> table, of the data type specified in the <b>coltype</b> column
<b>seclabelid</b>	INTEGER	The label ID of the security label associated with the column if it is a protected column. NULL otherwise.
<b>colattr</b>	SMALLINT	<p><b>HIDDEN</b></p> <p>1 - Hidden column</p> <p><b>ROWVER</b></p> <p>2 - Row version column</p> <p><b>ROW_CHKSUM</b></p> <p>4 - Row key column</p> <p><b>ER_CHECKVER</b></p> <p>8 - ER row version column</p> <p><b>UPGRD1_COL</b></p> <p>16 - ER auto primary key column</p> <p><b>UPGRD2_COL</b></p> <p>32 - ER auto primary key column</p> <p><b>UPGRD3_COL</b></p> <p>64 - ER auto primary key column</p> <p><b>PK_NOTNULL</b></p> <p>128 - NOT NULL by PRIMARY KEY</p>

**Note:**

<sup>1</sup> In DB-Access, an offset value of 256 is always added to these **coltype** codes because DB-Access sets SERIAL, SERIAL8, and BIGSERIAL columns to NOT NULL.

<sup>2</sup> 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 75](#) system catalog table.

<sup>3</sup> 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
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 23](#).

## 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:

$$\text{collength} = (\text{min\_space} * 256) + \text{max\_size}$$

- If the **collength** value is negative:

$$\text{collength} + 65536 = (\text{min\_space} * 256) + \text{max\_size}$$

### 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:

$$(\text{length} * 256) + (\text{first\_qualifier} * 16) + \text{last\_qualifier}$$

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

Field qualifier	Value	Field qualifier	Value
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 **sysindicies**. Because indexes can be shared, more than one constraint can be associated with an index. The **sysconstraints** table has the following columns.

**Table 11. SYSCONSTRAINTS table column descriptions**

Column	Type	Explanation
<b>constrid</b>	SERIAL	Code uniquely identifying the constraint
<b>constrname</b>	VARCHAR(128)	Name of the constraint
<b>owner</b>	VARCHAR(32)	Name of the owner of the constraint
<b>tabid</b>	INTEGER	Code uniquely identifying the table
<b>constrtype</b>	CHAR(1)	Code identifying the constraint type: <ul style="list-style-type: none"> <li>• C = Check constraint</li> <li>• N = Not NULL</li> <li>• P = Primary key</li> <li>• R = Referential</li> <li>• T = Table</li> <li>• U = Unique</li> </ul>
<b>idxname</b>	VARCHAR(128)	Name of index corresponding to constraint
<b>collation</b>	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 12. SYSDEFAULTS table column descriptions**

Column	Type	Explanation
<b>tabid</b>	INTEGER	Code uniquely identifying a table. When the <b>class</b> column contains the code P, then the <b>tabid</b> column references a procedure ID not a table ID.
<b>colno</b>	SMALLINT	Code uniquely identifying a column.
<b>type</b>	CHAR(1)	Code identifying the type of default value:  C = Current@ L = Literal value N = NULL S = Dbservername or Sitename T = Today U = User
<b>default</b>	CHAR(256)	If <b>sysdefaults.type</b> = L, a literal default value.
<b>class</b>	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 13. SYSDEPEND table column descriptions**

Column	Type	Explanation
<b>btabid</b>	INTEGER	Code uniquely identifying the base table or view
<b>btype</b>	CHAR(1)	Base object type: T = Table V = View
<b>dtabid</b>	INTEGER	Code uniquely identifying a dependent table or view
<b>dtype</b>	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 14. SYSDIRECTIVES table column descriptions**

Column	Type	Explanation
<b>id</b>	SERIAL	Unique code identifying the optimizer directive
<b>query</b>	TEXT	Text of the query as it exists in the application
<b>directives</b>	TEXT	Text of the optimizer directive, without comments
<b>directive_code</b>	BYTE	Encoded directive
<b>active</b>	SMALLINT	Integer code that identifies whether this entry is active ( = 1 ) or test only ( = 2 )
<b>hash_code</b>	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 15. SYSDISTRIB table column descriptions**

Column	Type	Explanation
<b>tabid</b>	INTEGER	Code identifying the table from which data values were gathered
<b>colno</b>	SMALLINT	Column number in the source table
<b>seqno</b>	INTEGER	Ordinal number for multiple entries
<b>constructed</b>	DATETIME YEAR TO FRACTION(5)	Date when the data distribution was created
<b>mode</b>	CHAR(1)	Optimization level: M = Medium H = High
<b>resolution</b>	SMALLFLOAT	Specified in the UPDATE STATISTICS statement
<b>confidence</b>	SMALLFLOAT	Specified in the UPDATE STATISTICS statement
<b>encdat</b>	STAT	Statistics information
<b>type</b>	CHAR(1)	Type of statistics: A = <b>encdat</b> has ASCII-encoded histogram in fixed-length character field S = <b>encdat</b> has user-defined statistics
<b>smplsize</b>	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.
<b>rowssmpld</b>	FLOAT	Number of rows in the sample
<b>constr_time</b>	DATETIME YEAR TO FRACTION(5)	Time when the distribution was recorded
<b>ustnrows</b>	FLOAT	Rows in fragment when distribution was calculated.
<b>ustbuildduration</b>	INTERVAL HOUR TO FRACTION(5)	Time spent calculating the distribution statistics for this column
<b>nupdates</b>	FLOAT	Number of updates to the table
<b>nDeletes</b>	FLOAT	Number of deletes to the table
<b>ninserts</b>	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 16. SYSDOMAINS table column descriptions**

Column	Type	Explanation
<b>id</b>	SERIAL	Unique code identifying the domain
<b>owner</b>	CHAR(32)	Name of the owner of the domain
<b>name</b>	VARCHAR(128)	Name of the domain
<b>type</b>	SMALLINT	Code identifying the type of domain

There is no index on this view.

## SYSEERRORS

The **syserrors** system catalog table stores information about error, warning, and informational messages returned by DataBlade® modules and user-defined routines using the **mi\_db\_error\_raise()** DataBlade® API function.

The **syserrors** table has the following columns.

Column	Type	Explanation
<b>sqlstate</b>	CHAR(5)	SQLSTATE value associated with the error.
<b>locale</b>	CHAR(36)	The locale with which this version of the message is associated (for example, <b>en_us.8859-1</b> )
<b>level</b>	SMALLINT	Reserved for future use
<b>seqno</b>	SMALLINT	Reserved for future use
<b>message</b>	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](#)

## SYSEXTCOLS

The **sysextcols** system catalog table contains a row that describes each of the internal columns in external table **tabid** of format type (**fmttype**) FIXED.

The **sysextcols** table has the following columns.

Column	Type	Explanation
<b>tabid</b>	INTEGER	Unique identifying code of a table
<b>colno</b>	SMALLINT	Code identifying the column
<b>exttype</b>	SMALLINT	Code identifying an external column type
<b>extstart</b>	SMALLINT	Starting position of column in the external data file
<b>extlength</b>	SMALLINT	External column length (in bytes)

Column	Type	Explanation
<b>nullstr</b>	CHAR(256)	Represents NULL in external data
<b>decprec</b>	SMALLINT	Precision for external decimals
<b>extstype</b>	VARCHAR(128,0)	External type name

No entries are stored in **sysextcols** for DELIMITED or HCL Informix® format external files.

You can use the DBSCHEMA utility to write out the description of the external tables. To query these system catalog tables about an external table, use the **tabid** as stored in **systables** with **tabtype** = 'E'.

An index on the **tabid** column allows duplicate values.

## SYSEXTDFILES

The **sysextfiles** system catalog table contains identifying codes and the paths of external tables.

For each external table, at least one row exists in the **sysextfiles** system catalog table, which has the following columns.

Column	Type	Explanation
<b>tabid</b>	INTEGER	Unique identifying code of an external table
<b>dfentry</b>	CHAR(469)	Absolute source or target file path
<b>blobdir</b>	CHAR(344)	Absolute or relative directory name
<b>clobdir</b>	CHAR(344)	Absolute or relative directory name

You can use DBSCHEMA to write out the description of the external tables. To query these system catalog tables about an external table, use the **tabid** as stored in **systables** with **tabtype** = 'E'.

An index on the **tabid** column allows duplicate values.

## SYSEXTERNAL

For each external table, a single row exists in the **sysextexternal** system catalog table.

The **tabid** column associates the external table record in this system catalog table with an entry in **systables**.

Column	Type	Explanation
<b>tabid</b>	INTEGER	Unique identifying code of an external table
<b>fmttype</b>	CHAR(1)	Type of format: D = (delimited) F = (fixed) I = (HCL Informix®)
<b>codeset</b>	VARCHAR(128)	Reserved for future use
<b>recdelim</b>	VARCHAR(128)	The record delimiter
<b>flddelim</b>	CHAR(4)	The field delimiter

Column	Type	Explanation
<b>datefmt</b>	CHAR(8)	Reserved for future use
<b>moneyfmt</b>	CHAR(20)	Reserved for future use
<b>maxerrors</b>	INTEGER	Number of errors to allow
<b>rejectfile</b>	CHAR(464)	Name of the reject file
<b>flags</b>	INTEGER	Optional <b>load</b> flags
<b>ndfiles</b>	INTEGER	Number of data files in <b>sysextdfiles</b>

You can use the **dbschema** utility to write out the description of the external tables. To query these system catalog tables about an external table, use the **tabid** as stored in **systable**s with **tabtype** = 'E'.

An index on the **tabid** column allows only unique values.

## SYSFRAGAUTH

The **sysfragauth** system catalog table stores information about the privileges that are granted on table fragments. This table has the following columns.

**Table 17. SYSFRAGAUTH table column descriptions**

Column	Type	Explanation
<b>grantor</b>	CHAR(32)	Name of the grantor of privilege
<b>grantee</b>	CHAR(32)	Name of the grantee of privilege
<b>tabid</b>	INTEGER	Identifying code of the fragmented table
<b>fragment</b>	VARCHAR(128)	Name of dbspace where fragment is stored
<b>fragauth</b>	CHAR(6)	A 6-byte pattern specifying fragment privileges (including 3 bytes reserved for future use): <ul style="list-style-type: none"> <li>• u or U = Update</li> <li>• i or I = Insert</li> <li>• d or D = Delete</li> </ul>

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--

## SYSFRAGDIST

The **sysfragdist** system catalog table stores fragment-level column statistics for fragmented tables and indexes. One row exists for each table fragment or index fragment.

Only columns in fragmented tables are described here. (For table-level column statistics, see the **sysdistrib** system catalog table.)

The **sysfragdist** table has the following columns.

Column	Type	Explanation
<b>tabid</b>	INTEGER	Unique identifying code of table (= <b>sysables.tabid</b> )
<b>fragid</b>	INTEGER	Unique identifying code of fragment (= <b>sysfragments.partnum</b> )
<b>colno</b>	SMALLINT	Unique identifying code of column (= <b>syscolumns.colno</b> )
<b>seqno</b>	SMALLINT	Sequence number (for distributions that span multiple rows)
<b>mode</b>	CHAR(1)	UPDATE STATISTICS mode (H = high, or M = medium)
<b>resolution</b>	SMALLFLOAT	Average percentage of the sample in each bin
<b>confidence</b>	SMALLFLOAT	Estimated likelihood that a MEDIUM mode sample value is equivalent to an exact HIGH mode result
<b>rowssampled</b>	FLOAT	Number of rows in the sample
<b>ustbuildduration</b>	INTERVAL HOUR TO FRACTION(5)	Time spent to calculate the distribution for this column
<b>constr_time</b>	DATETIME YEAR TO FRACTION(5)	Time when the distribution was recorded
<b>ustnrows</b>	FLOAT	Rows in fragment when distribution was calculated.

Column	Type	Explanation
<b>minibinsize</b>	FLOAT	For internal use only
<b>nupdates</b>	FLOAT	Number of updates to the table
<b>ndeletes</b>	FLOAT	Number of deletes to the table
<b>ninserts</b>	FLOAT	Number of inserts to the table
<b>version</b>	INTEGER	Reserved for future use
<b>dbsnum</b>	INTEGER	Unique identifying code of sbospace where <b>encdist</b> is stored
<b>encdist</b>	STAT	Encrypted fragment distribution

The set of rows with a given combination of **tabid**, **fragid**, and **colno** values identifies the column statistics for that fragment of a table. These statistics can span multiple rows by using the **seqno** column for sequence numbering.

The *mode*, *resolution* and *confidence* values that are specified in the UPDATE STATISTICS MEDIUM or HIGH statement that calculate the column statistics for the fragment are recorded in the **sysfragdist** columns of the same names. To use existing fragment statistics to build table statistics, these three parameters should not change between UPDATE STATISTICS statements that reference the fragments of the same table. The only exception to this is that "H" mode fragmented statistics can be used to build "M" mode table statistics.

Column distribution statistics for the fragment are stored in the column **encdist**. The **dbsnum** column stores the identifying code of the smart blob space where the **encdist** object describing this fragment is stored. By default, the SBSPACENAME configuration parameter setting is the identifier of the sbospace whose identifying code is in the **dbsnum** column.

The following three columns record counts of how many DML operations modifying data rows were performed on the fragment 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 existed when distribution statistics were generated. When UPDATE STATISTICS runs in MEDIUM or HIGH mode against the fragmented table with fragment level statistics, the database server compares the stored values in these columns with the current values in the partition.

When 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 for the fragment. Column statistics for the fragment corresponding to the row in the **sysfragdist** table are not updated if the

sum of the stored values differs from the sum of these current DML counter values for the partition page by less than the threshold specified by the setting of the STATCHANGE table attribute or of the STATCHANGE configuration parameter.

## 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
<b>fragtype</b>	CHAR(1)	Code indicating the type of fragmented object: <ul style="list-style-type: none"> <li>• I = Original index fragment</li> <li>• T = Original table fragment</li> </ul>
<b>tabid</b>	INTEGER	Unique identifying code of table
<b>indexname</b>	VARCHAR(128)	Name of index
<b>colno</b>	INTEGER	Identifying code of TEXT or BYTE column, or the upper limit on the number of rolling window fragments
<b>partn</b>	INTEGER	Identifying code of physical storage location
<b>strategy</b>	CHAR(1)	Code for type of fragment distribution strategy: <ul style="list-style-type: none"> <li>• R = Round-robin distribution strategy</li> <li>• E = Expression-based distribution strategy</li> <li>• I = IN DBSPACE clause specifies a storage location as part of distribution strategy</li> <li>• N = raNge-iNterval (or rolliNg wiNdoW) distribution strategy</li> <li>• N = raNge-iNterval distribution strategy</li> <li>• L = List distribution strategy</li> <li>• T = Table-based distribution strategy</li> <li>• H = table is a subtable within a table Hierarchy</li> </ul>
<b>location</b>	CHAR(1)	Reserved for future use; shows L for local
<b>servername</b>	VARCHAR(128)	Reserved for future use
<b>evalpos</b>	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 <b>expertext</b> field:

Column	Type	Explanation
		<ul style="list-style-type: none"> <li>• -1 = List of dbspaces for interval fragments</li> <li>• -2 = Interval value</li> <li>• -3 = Fragmentation key</li> <li>• -4 = Rolling window fragment</li> </ul> <p>Fragmentation by LIST also uses the -3 value.</p>
<b>exprtext</b>	TEXT	<p>Expression for fragmentation strategy</p> <p>For fragmentation by INTERVAL, LIST, or rolling window, provides the information corresponding to the value of the <b>evalpos</b> field.</p> <p>For fragmentation by INTERVAL or LIST, provides the information corresponding to the value of the <b>evalpos</b> field.</p>
<b>exprbin</b>	BYTE	Binary version of expression
<b>exprarr</b>	BYTE	Range-partitioning data to optimize expression in range-expression fragmentation strategy
<b>flags</b>	INTEGER	Used internally
<b>dbspace</b>	VARCHAR(128)	Name of dbspace storing this fragment
<b>levels</b>	SMALLINT	Number of B-tree index levels
<b>npused</b>	FLOAT	<p>For table-fragmentation strategies: the number of data pages</p> <p>For index-fragmentation strategies: the number of leaf pages</p> <p>For rolling window tables: the units for the storage size limit in nrows</p>
<b>nrows</b>	FLOAT	<p>For tables: the number of rows in the fragment.</p> <p>For indexes: the number of unique keys.</p> <p>For rolling window tables: the upper limit on storage size in the purge policy.</p>
<b>clust</b>	FLOAT	Degree of index clustering; smaller numbers correspond to greater clustering.
<b>partition</b>	VARCHAR(128)	Fragment name. This can match the name of the dbspace that stores the fragment, or can be an arbitrary name.
<b>version</b>	SMALLINT	Number that increments when fragment statistics is updated
<b>nupdates</b>	FLOAT	Number of updates to the fragment

Column	Type	Explanation
<b>ndeletes</b>	FLOAT	Number of deletes to the fragment
<b>ninserts</b>	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 31](#) 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 18. SYSINDEXES table column descriptions**

Column	Type	Explanation
<b>idxname</b>	VARCHAR(128)	Index name
<b>owner</b>	VARCHAR(32)	Owner of index (user <b>informix</b> for system catalog tables and <i>username</i> for database tables)
<b>tabid</b>	INTEGER	Unique identifying code of table
<b>idxtype</b>	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

**Table 18. SYSINDEXES table column descriptions**

(continued)

Column	Type	Explanation
<b>clustered</b>	CHAR(1)	Clustered or nonclustered index (C = Clustered)
<b>part1</b>	SMALLINT	Column number ( <b>colno</b> ) of a single index or the 1st component of a composite index
<b>part2</b>	SMALLINT	2nd component of a composite index
<b>part3</b>	SMALLINT	3rd component of a composite index
<b>part4</b>	SMALLINT	4th component of a composite index
<b>part5</b>	SMALLINT	5th component of a composite index
<b>part6</b>	SMALLINT	6th component of a composite index
<b>part7</b>	SMALLINT	7th component of a composite index
<b>part8</b>	SMALLINT	8th component of a composite index
<b>part9</b>	SMALLINT	9th component of a composite index
<b>part10</b>	SMALLINT	10th component of a composite index
<b>part11</b>	SMALLINT	11th component of a composite index
<b>part12</b>	SMALLINT	12th component of a composite index
<b>part13</b>	SMALLINT	13th component of a composite index
<b>part14</b>	SMALLINT	14th component of a composite index
<b>part15</b>	SMALLINT	15th component of a composite index
<b>part16</b>	SMALLINT	16th component of a composite index
<b>levels</b>	SMALLINT	Number of B-tree levels
<b>leaves</b>	INTEGER	Number of leaves
<b>nunique</b>	INTEGER	Number of unique keys in the first column
<b>clust</b>	INTEGER	Degree of clustering; smaller numbers correspond to greater clustering
<b>idxflags</b>	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 19. sysindices system catalog table columns**

Col umn	Type	Explanation
<b>idxn ame</b>	VARCHAR(1 28)	Name of index
<b>ow ner</b>	VARCHAR( 32)	Name of owner of index (user <b>informix</b> for system catalog tables and <i>username</i> for database tables)
<b>tabid</b>	INTEGER	Unique identifying code of table
<b>idxt ype</b>	CHAR(1)	Uniqueness status  U = Unique values required D = Duplicates allowed
<b>clust ered</b>	CHAR(1)	Clustered or nonclustered status (C = Clustered)
<b>lev els</b>	SMALLINT	Number of tree levels
<b>lea ves</b>	FLOAT	Number of leaves
<b>nuni que</b>	FLOAT	Number of unique keys in the first column
<b>clust</b>	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.
<b>nr ows</b>	FLOAT	Estimated number of rows in the table (zero until UPDATE STATISTICS is run on the table)

**Table 19. sysindices system catalog table columns**

(continued)

Column	Type	Explanation
<b>index keys</b>	INDEXKEYA RRAY	Internal representation of the index keys. Column can have up to three fields, in the format: <b>procid</b> , ( <i>col1,col2, . . . , coln</i> ), <b>opclassid</b> where $1 < n < 341$
<b>amid</b>	INTEGER	Unique identifying code of the access method that implements this index. (Value = <b>am_id</b> for that access method in the <b>sysams</b> table.)
<b>ampa ram</b>	LVARCHAR(2048)	List of parameters used to customize the <b>amid</b> access method behavior
<b>collation</b>	CHAR(32)	Database locale whose collating order was in effect at the time of index creation
<b>page size</b>	INTEGER	Size of the page, in bytes, where this index is stored
<b>nhashcols</b>	SMALLINT	Number of hashed columns in a FOT index
<b>nbuckets</b>	SMALLINT	Number of subtrees (buckets) in a forest of trees (FOT) index
<b>ustlastwts</b>	DATETIME YEAR TO FRACTION	Date and time when index statistics were last recorded
<b>ustbuildduration</b>	INTERVAL HOUR TO FRACTION (5)	Time required to calculate index statistics
<b>nupdates</b>	FLOAT	Number of updates to the table
<b>ndeletes</b>	FLOAT	Number of deletes to the table
<b>number of inserts</b>	FLOAT	Number of inserts to the table
<b>first extent size</b>	INT	Size (in KB) of the first extent of the index
<b>next extent size</b>	INT	Size (in KB) of the next extent of the index



Table 19. **sysindices** system catalog table columns

(continued)

Column	Type	Explanation
<b>index</b>	INT	
<b>attr</b>		<ul style="list-style-type: none"> <li>• 0x00000001 = The index has a partial column key</li> <li>• 0x00000002 = The index is compressed</li> <li>• 0x00000004 = The index is on a BSON column</li> </ul>
<b>jpa</b>	LVARCHAR(	BSON index information
<b>ram</b>	2048)	



**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 41](#).

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 31](#) system catalog table.

The **fectsize** 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 **fectsize** and **nextsize** column values are set to 0. When the database server is converted from a release earlier than Version 11.70, the **fectsize** 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.

## 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
<b>child</b>	INTEGER	Identifying code of the subtable or subtype
<b>parent</b>	INTEGER	Identifying code of the supertable or supertype
<b>class</b>	CHAR(1)	Inheritance class: <b>U</b> = named ROW type <b>T</b> = table

The **child** and **parent** values are from **sysxdtypes.extended\_id** for named ROW types, or from **sysables.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 20. SYSLANGAUTH table column descriptions**

Column	Type	Explanation
<b>grantor</b>	VARCHAR(32)	Name of the grantor of the language authorization
<b>grantee</b>	VARCHAR(32)	Name of the grantee of the language authorization
<b>langid</b>	INTEGER	Identifying code of language in <b>sysroutinelangs</b> table
<b>langauth</b>	CHAR(1)	The language authorization:  <ul style="list-style-type: none"> <li>u = Usage privilege granted</li> <li>U = Usage privilege granted WITH GRANT OPTION</li> </ul>

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 21. SYSLOGMAP table column descriptions**

Column	Type	Explanation
<b>tabloc</b>	INTEGER	Code for the location of a table in another database
<b>tabid</b>	INTEGER	Unique identifying code of the table
<b>fragid</b>	INTEGER	Identifying code of the fragment
<b>flags</b>	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 22. SYSOBJSTATE table column descriptions**

Column	Type	Explanation
<b>objtype</b>	CHAR(1)	Code for the type of database object: <ul style="list-style-type: none"> <li>• C = Constraint</li> <li>• I = Index</li> <li>• T = Trigger</li> </ul>
<b>owner</b>	VARCHAR(32)	Authorization identifier of the owner of the database object
<b>name</b>	VARCHAR(128)	Name of the database object
<b>tabid</b>	INTEGER	Identifying code of table on which the object is defined

**Table 22. SYSOBJSTATE table column descriptions**

(continued)

Col umn	Type	Explanation
<b>state</b>	CHAR(1)	The current state (object mode) of the database object. This value can be one of the following codes: <ul style="list-style-type: none"> <li>• D = Disabled</li> <li>• E = Enabled</li> <li>• F = Filtering with no integrity-violation errors</li> <li>• G = Filtering with integrity-violation error</li> </ul>

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.

## 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
<b>opclassname</b>	VARCHAR(128)	Name of the operator class
<b>owner</b>	VARCHAR(32)	Name of the owner of the operator class
<b>amid</b>	INTEGER	Identifying code of the secondary access method associated with this operator class
<b>opclassid</b>	SERIAL	Identifying code of the operator class
<b>ops</b>	LVARCHAR(2048)	List of names of the operators that belong to this operator class
<b>support</b>	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.

## SYSOPCLSTR

The **sysopclstr** system catalog table defines each optical cluster in the database. The table contains one row for each optical cluster.

The **sysopclstr** table has the following columns.

Column	Type	Explanation
<b>owner</b>	VARCHAR(32)	Name of the owner of the optical cluster
<b>clstrname</b>	VARCHAR(128)	Name of the optical cluster
<b>clstrsize</b>	INTEGER	Size of the optical cluster
<b>tabid</b>	INTEGER	Unique identifying code for the table
<b>blobcol1</b>	SMALLINT	BYTE or TEXT column number 1
<b>blobcol2</b>	SMALLINT	BYTE or TEXT column number 2
<b>blobcol3</b>	SMALLINT	BYTE or TEXT column number 3
<b>blobcol4</b>	SMALLINT	BYTE or TEXT column number 4
<b>blobcol5</b>	SMALLINT	BYTE or TEXT column number 5
<b>blobcol6</b>	SMALLINT	BYTE or TEXT column number 6
<b>blobcol7</b>	SMALLINT	BYTE or TEXT column number 7
<b>blobcol8</b>	SMALLINT	BYTE or TEXT column number 8
<b>blobcol9</b>	SMALLINT	BYTE or TEXT column number 9
<b>blobcol10</b>	SMALLINT	BYTE or TEXT column number 10
<b>blobcol11</b>	SMALLINT	BYTE or TEXT column number 11
<b>blobcol12</b>	SMALLINT	BYTE or TEXT column number 12
<b>blobcol13</b>	SMALLINT	BYTE or TEXT column number 13
<b>blobcol14</b>	SMALLINT	BYTE or TEXT column number 14
<b>blobcol15</b>	SMALLINT	BYTE or TEXT column number 15
<b>blobcol16</b>	SMALLINT	BYTE or TEXT column number 16
<b>clstrkey1</b>	SMALLINT	Cluster key number 1
<b>clstrkey2</b>	SMALLINT	Cluster key number 2
<b>clstrkey3</b>	SMALLINT	Cluster key number 3
<b>clstrkey4</b>	SMALLINT	Cluster key number 4
<b>clstrkey5</b>	SMALLINT	Cluster key number 5
<b>clstrkey6</b>	SMALLINT	Cluster key number 6
<b>clstrkey7</b>	SMALLINT	Cluster key number 7

Column	Type	Explanation
<b>clstrkey8</b>	SMALLINT	Cluster key number 8
<b>clstrkey9</b>	SMALLINT	Cluster key number 9
<b>clstrkey10</b>	SMALLINT	Cluster key number 10
<b>clstrkey11</b>	SMALLINT	Cluster key number 11
<b>clstrkey12</b>	SMALLINT	Cluster key number 12
<b>clstrkey13</b>	SMALLINT	Cluster key number 13
<b>clstrkey14</b>	SMALLINT	Cluster key number 14
<b>clstrkey15</b>	SMALLINT	Cluster key number 15
<b>clstrkey16</b>	SMALLINT	Cluster key number 16

The contents of this table are sensitive to CREATE OPTICAL CLUSTER, ALTER OPTICAL CLUSTER, and DROP OPTICAL CLUSTER statements that have been executed on databases that support optical cluster subsystems. Changes that affect existing optical clusters are reflected in this table only after you run the UPDATE STATISTICS statement.

A composite index on the **clstrname** and **owner** columns allows only unique values. A simple index on the **tabid** column allows duplicate 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 23. SYSPROCAUTH table column descriptions**

Column	Type	Explanation
<b>grantor</b>	VARCHAR(32)	Name of grantor of privileges to access the routine
<b>grantee</b>	VARCHAR(32)	Name of grantee of privileges to access the routine
<b>procid</b>	INTEGER	Unique identifying code of the routine
<b>procauth</b>	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 24. SYSPROCBODY table column descriptions**

Column	Type	Explanation
<b>procid</b>	INTEGER	Unique identifying code for the routine
<b>datakey</b>	CHAR(1)	Type of information in the <b>data</b> 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
<b>seqno</b>	INTEGER	Line number within the routine
<b>data</b>	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 25. SYSPROCCOLUMNS table column descriptions**

Column	Type	Explanation
<b>procid</b>	INTEGER	Unique identifying code of the routine
<b>paramid</b>	INTEGER	Unique identifying code of the parameter
<b>paramname</b>	VARCHAR (IDENTSIZE)	Name of the parameter
<b>paramtype</b>	SMALLINT	Identifies the type of parameter
<b>paramlen</b>	SMALLINT	Specifies the length of the parameter
<b>paramxid</b>	INTEGER	Specifies the extended type ID for the parameter
<b>paramattr</b>	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 26. SYSPROCEDURES table column descriptions**

Column	Type	Explanation
<b>procname</b>	VARCHAR(128)	Name of routine
<b>owner</b>	VARCHAR(32)	Name of owner
<b>procid</b>	SERIAL	Unique identifying code for the routine
<b>mode</b>	CHAR(1)	Mode type:  D or d = DBA O or o = Owner



Table 26. SYSPROCEDURES table column descriptions (continued)

Column	Type	Explanation
		P or p = Protected R or r = Restricted T or t = Trigger
<b>retsize</b>	INTEGER	Compiled size (in bytes) of returned values
<b>symsize</b>	INTEGER	Compiled size (in bytes) of symbol table
<b>datasize</b>	INTEGER	Compiled size (in bytes) of constant data
<b>codesize</b>	INTEGER	Compiled size (in bytes) of routine code
<b>numargs</b>	INTEGER	Number of arguments to routine
<b>isproc</b>	CHAR(1)	Specifies if the routine is a procedure or a function:  t = procedure f = function
<b>specificname</b>	VARCHAR(128)	Specific name for the routine
<b>externalname</b>	VARCHAR(255)	Location of the external routine. This item is language-specific in content and format.
<b>paramstyle</b>	CHAR(1)	Parameter style: i = HCL Informix®
<b>langid</b>	INTEGER	Language code (in <b>sysroutinelangs</b> table)
<b>paramtypes</b>	RTNPARAMTYPES	Information describing the parameters of the routine
<b>variant</b>	BOOLEAN	Whether the routine is VARIANT or not:  t = is VARIANT f = is not VARIANT
<b>client</b>	BOOLEAN	Reserved for future use
<b>handlesnulls</b>	BOOLEAN	NULL handling indicator:  t = handles NULLs f = does not handle NULLs
<b>percallcost</b>	INTEGER	Amount of CPU per call  Integer cost to execute UDR: cost/call - 0 -(2^31-1)

**Table 26. SYSPROCEDURES table column descriptions (continued)**

Column	Type	Explanation
<b>commutator</b>	VARCHAR(128)	Name of commutator function
<b>negator</b>	VARCHAR(128)	Name of the negator function
<b>selfunc</b>	VARCHAR(128)	Name of function to estimate selectivity of the UDR
<b>internal</b>	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
<b>class</b>	CHAR(18)	CPU class by which the routine should be executed
<b>stack</b>	INTEGER	Stack size in bytes required per invocation
<b>parallelizable</b>	BOOLEAN	Parallelization indicator for UDR:  t = parallelizable f = not parallelizable
<b>costfunc</b>	VARCHAR(128)	Name of the cost function for the UDR
<b>selconst</b>	SMALLFLOAT	Selectivity constant for UDR
<b>procflags</b>	INTEGER	For internal use only
<b>collation</b>	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 27. SYSPROCPLAN table column descriptions**

Column	Type	Explanation
<b>procid</b>	INTEGER	Identifying code for the routine
<b>planid</b>	INTEGER	Identifying code for the plan
<b>datakey</b>	CHAR(1)	Type of information stored in <b>data</b> column:  D = Dependency list I = Information record Q = Execution plan
<b>seqno</b>	INTEGER	Line number within the plan
<b>created</b>	DATE	Date when plan was created
<b>datasize</b>	INTEGER	Size (in bytes) of the list or plan
<b>data</b>	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 28. SYSREFERENCES table column descriptions**

Column	Type	Explanation
<b>constrid</b>	INTEGER	Code uniquely identifying the constraint
<b>primary</b>	INTEGER	Identifying code of the corresponding primary key
<b>ptabid</b>	INTEGER	Identifying code of the table that is the primary key
<b>updrule</b>	CHAR(1)	Reserved for future use; displays an R
<b>delrule</b>	CHAR(1)	Whether constraint uses cascading delete or restrict rule:  C = Cascading delete R = Restrict (default)
<b>matchtype</b>	CHAR(1)	Reserved for future use; displays an N
<b>pendant</b>	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 29. SYSROLEAUTH table column descriptions**

Column	Type	Explanation
<b>rolename</b>	VARCHAR(32)	Name of the role

Table 29. SYSROLEAUTH table column descriptions

(continued)

Column	Type	Explanation
<b>grantee</b>	VARCHAR(32)	Name of the grantee of the role
<b>is_grantable</b>	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
<b>langid</b>	SERIAL	Code uniquely identifying a supported language
<b>langname</b>	CHAR(30)	Name of the language, such as C or SPL
<b>langinitfunc</b>	VARCHAR(128)	Name of initialization function for the language
<b>langpath</b>	CHAR(255)	Directory path for the UDR language
<b>langclass</b>	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
<b>GRANTEE</b>	CHAR(32)	The name of the label grantee
<b>secpolicyid</b>	INTEGER	The ID of the security policy to which the security label belongs.

Column	Type	Explanation
<b>readseclabelid</b>	INTEGER	The security label ID of the security label granted for read access
<b>writeseclabelid</b>	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
<b>compname</b>	VARCHAR(128)	Component name
<b>compid</b>	SERIAL	Component ID
<b>comptype</b>	CHAR(1)	The component type:  A = array S = set T = tree
<b>numelements</b>	INTEGER	Number of elements in the component
<b>coveringinfo</b>	VARCHAR(128)	Internal encoding information
<b>numalters</b>	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
<b>compid</b>	INTEGER	Component ID
<b>element</b>	VARCHAR(32)	Element name
<b>elementencoding</b>	CHAR(8)	Encoded form of the element
<b>parentelement</b>	VARCHAR(32)	The name of the parent elements for tree components. The value is NULL for the following items:

Column	Type	Explanation
		Set components Array components Root nodes of a tree component
<b>alterversion</b>	SMALLINT	The number of the alter operation when the element is added. This value is used by the <b>dbexport</b> and <b>dbimport</b> commands.

## SYSSECLABELNAMES

The **sysseclabelnames** system catalog table records the security label names. It has these columns.

Column	Type	Explanation
<b>secpolicyid</b>	INTEGER	The ID of the security policy to which the security label belongs.
<b>seclabelname</b>	VARCHAR(128)	The name of the security label
<b>seclabelid</b>	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
<b>secpolicyid</b>	INTEGER	ID of the security policy to which the security label belongs
<b>seclabelid</b>	INTEGER	Security label ID
<b>sysseclabelnames</b>	VARCHAR(128)	Security label encoding

## SYSSECPOLICIES

The **syssecpolicies** system catalog table records security policies It has these columns.

Column	Type	Explanation
<b>secpolicyname</b>	VARCHAR(128)	Security policy name
<b>secpolicyid</b>	SERIAL	Security policy ID
<b>numcomps</b>	SMALLINT	Number of security label components in the security policy
<b>comptypelist</b>	CHAR(16)	An ordered list of the type of each component in the policy.

Column	Type	Explanation
		<p>A = array                      S = set                      T = tree                      - = Beyond NUMCOMPS</p>
<b>overrideseclabel</b>	CHAR(1)	<p>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.</p> <ul style="list-style-type: none"> <li>• Y: The security label provided is ignored and replaced by the user's security label for write access.</li> <li>• N: Return an error when not authorized to write a security label.</li> </ul>

## SYSSECPOLICYCOMPONENTS

The **syssecpolicycomponents** system catalog table records the components for each security policies. It has these columns.

Column	Type	Explanation
<b>secpolicyid</b>	INTEGER	Security policy ID
<b>compid</b>	INTEGER	ID of a component of the label security policy
<b>compno</b>	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
<b>grantee</b>	CHAR(32)	The user who has this exemption
<b>secpolicyid</b>	INTEGER	ID of the policy on which the exemption is granted
<b>exemption</b>	CHAR(6)	The exemption given to the user who is identified in the GRANTEE column. The six characters have the following meanings:



Column	Type	Explanation
		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
<b>seqid</b>	SERIAL	Code uniquely identifying the sequence object
<b>tabid</b>	INTEGER	Identifying code of the sequence as a table object
<b>start_val</b>	INT8	Starting value of the sequence
<b>inc_val</b>	INT8	Value of the increment between successive values
<b>max_val</b>	INT8	Largest possible value of the sequence
<b>min_val</b>	INT8	Smallest possible value of the sequence
<b>cycle</b>	CHAR(1)	Zero means NOCYCLE, 1 means CYCLE
<b>restart_val</b>	INT8	Starting value of the sequence after ALTER SEQUENCE RESTART was run
<b>cache</b>	INTEGER	Number of preallocated values in sequence cache
<b>order</b>	CHAR(1)	Zero means NOORDER, 1 means ORDER

## SYSSURROGATEAUTH

The **sys surrogateauth** 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 30. SYSSURROGATEAUTH table column descriptions**

Column	Type	Explanation
<b>trusteduser</b>	CHAR(32)	Trusted user name or role.
<b>surrogateuser</b>	CHAR(32)	Surrogate user name.

## SYSSYNONYMS

The **syssynonyms** system catalog table is unused. The **syssynname** table describes synonyms. The **syssynonyms** system catalog table has the following columns.

**Table 31. SYSSYNONYMS table column descriptions**

Column	Type	Explanation
<b>owner</b>	VARCHAR(32)	Name of the owner of the synonym
<b>synname</b>	VARCHAR(128)	Name of the synonym
<b>created</b>	DATE	Date when the synonym was created
<b>tabid</b>	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
<b>tabid</b>	INTEGER	Identifying code of the public synonym
<b>servername</b>	VARCHAR(128)	Name of an external database server
<b>dbname</b>	VARCHAR(128)	Name of an external database
<b>owner</b>	VARCHAR(32)	Name of the owner of an external object
<b>tablename</b>	VARCHAR(128)	Name of an external table or view
<b>btbid</b>	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 **btbid** columns are used. If you define a synonym for a table that is external to your current database, the **btbid** column is not used, but the **tabid**, **servername**, **dbname**, **owner**, and **tablename** 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 **btbid** 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 32. SYSTABAMDATA table column descriptions**

Column	Type	Explanation
<b>tabid</b>	INTEGER	Identifying code of the table
<b>am_param</b>	LVARCHAR(8192)	Access method parameter choices

**Table 32. SYSTABAMDATA table column descriptions**

(continued)

Column	Type	Explanation
<b>am_space</b>	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 33. SYSTABAUTH table column descriptions**

Column	Type	Explanation
<b>grantor</b>	VARCHAR(32)	Name of the grantor of privilege
<b>grantee</b>	VARCHAR(32)	Name of the grantee of privilege
<b>tabid</b>	INTEGER	Value from <b>systables.tabid</b> for database object
<b>tabauth</b>	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 41](#)). (In DB-Access, 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 **sysables** 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 34. SYSTABLES table column descriptions**

Column	Type	Explanation
<b>tablename</b>	VARCHAR(128)	Name of table, view, synonym, or sequence
<b>owner</b>	CHAR(32)	Owner of table (user <b>informix</b> for system catalog tables and <i>username</i> for database tables)
<b>partnum</b>	INTEGER	Physical storage location code
<b>tabid</b>	SERIAL	System-assigned sequential identifying number
<b>rowsize</b>	SMALLINT	Maximum row size in bytes ( < 32,768)
<b>ncols</b>	SMALLINT	Number of columns in the table
<b>nindexes</b>	SMALLINT	Number of indexes on the table
<b>nrows</b>	FLOAT	Number of rows in the table
<b>created</b>	DATE	Date when table was created or last modified
<b>version</b>	INTEGER	Number that changes when table is altered
<b>tabtype</b>	CHAR(1)	Code indicating the type of table object: <ul style="list-style-type: none"> <li>• T = Table</li> <li>• E = External Table</li> <li>• V = View</li> <li>• Q = Sequence</li> <li>• P = Private synonym</li> <li>• S = Public synonym</li> </ul> (Type S is unavailable in an ANSI-compliant database.)
<b>locklevel</b>	CHAR(1)	Lock mode for the table:

**Table 34. SYSTABLES table column descriptions (continued)**

Column	Type	Explanation
		<ul style="list-style-type: none"> <li>• B = Page and row level</li> <li>• P = Page level</li> <li>• R = Row level</li> </ul>
<b>npused</b>	FLOAT	Number of data pages that have ever been initialized in the tablespace by the database server
<b>fextsize</b>	INTEGER	Size of initial extent (in KB)
<b>nextsize</b>	INTEGER	Size of all subsequent extents (in KB)
<b>flags</b>	SMALLINT	<p>Codes for classifying permanent tables:</p> <p><b>ROWID</b></p> <p>1 - Has rowid column defined</p> <p><b>UNDER</b></p> <p>2 - Table created under a supertable</p> <p><b>VIEWREMOTE</b></p> <p>4 - View is based on a remote table</p> <p><b>CDR</b></p> <p>8 - Has CDRCOLS defined</p> <p><b>RAW</b></p> <p>16 - (Informix®) RAW table</p> <p><b>EXTERNAL</b></p> <p>32- External table</p> <p><b>AUDIT</b></p> <p>64 - Audit table attribute - FGA</p> <p><b>AQT</b></p> <p>128 - View is an AQT for DWA offloading</p> <p><b>VIRTAQT</b></p> <p>256 - View is a virtual AQT</p>
<b>site</b>	VARCHAR(128)	Reserved for future use
<b>dbname</b>	VARCHAR(128)	Reserved for future use

Table 34. SYSTABLES table column descriptions (continued)

Column	Type	Explanation
<b>type_xid</b>	INTEGER	Code from <b>sysxdtypes.extended_id</b> for typed tables, or 0 for untyped tables
<b>am_id</b>	INTEGER	Access method code (key to <b>sysams</b> table)  NULL or 0 indicates built-in storage manager
<b>pagesize</b>	INTEGER	The pagesize, in bytes, of the dbspace (or dbspaces, if the table is fragmented) where the table data resides.
<b>ustlowts</b>	DATETIME YEAR TO FRACTION (5)	When table, row, and page-count statistics were last recorded
<b>secpolicyid</b>	INTEGER	ID of the SECURITY policy attached to the table. NULL for non-protected tables
<b>protgranularity</b>	CHAR(1)	LBAC granularity level: <ul style="list-style-type: none"> <li>• R: Row level granularity</li> <li>• C: Column level granularity</li> <li>• B: Both column and row granularity</li> <li>• Blank for non-protected tables</li> </ul>
<b>statlevel</b>	CHAR(1)	Statistics level <ul style="list-style-type: none"> <li>• T = table</li> <li>• F = fragment</li> <li>• A = automatic</li> </ul>
<b>statchange</b>	SMALLINT	For internal use only

Each table, view, sequence, and synonym recorded in the **sysables** 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 **tablename** and **owner** columns also requires unique values.

The version column contains an encoded number that is stored in **sysables** 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 35. SYSTRACECLASSES table column descriptions**

Column	Type	Explanation
<b>name</b>	CHAR(18)	Name of the class of trace messages
<b>classid</b>	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 DataBlade® modules and user-defined routines. Developers use the tracing facility by calling the appropriate DataBlade® API 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 36. SYSTRACEMSGS table column descriptions**

Column	Type	Explanation
<b>name</b>	VARCHAR(128)	Name of the message
<b>msgid</b>	SERIAL	Identifying code of the message template
<b>locale</b>	CHAR(36)	Locale with which this version of the message is associated (for example, <b>en_us.8859-1</b> )
<b>seqno</b>	SMALLINT	Reserved for future use
<b>message</b>	VARCHAR(255)	The message text

DataBlade® 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 DataBlade® API 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 37. SYSTRIGBODY table column descriptions**

Column	Type	Explanation
<b>trigid</b>	INTEGER	Identifying code of the trigger
<b>datakey</b>	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
<b>seqno</b>	INTEGER	Page number of this data segment
<b>data</b>	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 38. SYSTRIGGERS table column descriptions**

Column	Type	Explanation
<b>trigid</b>	SERIAL	Identifying code of the trigger
<b>trigname</b>	VARCHAR(128)	Name of the trigger
<b>owner</b>	VARCHAR(32)	Name of the owner of the trigger
<b>tabid</b>	INTEGER	Identifying code of the triggering table
<b>event</b>	CHAR(1)	Code for the type of triggering event:  D = Delete trigger I = Insert trigger U = Update trigger S = Select trigger

**Table 38. SYSTRIGGERS table column descriptions (continued)**

Column	Type	Explanation
		d = INSTEAD OF Delete trigger i = INSTEAD OF Insert trigger u = INSTEAD OF Update trigger
<b>old</b>	VARCHAR(128)	Name of value before update
<b>new</b>	VARCHAR(128)	Name of value after update
<b>mode</b>	CHAR(1)	Reserved for future use
<b>collation</b>	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 39. SYSUSERS table column descriptions**

Column	Type	Explanation
<b>username</b>	VARCHAR(32)	Name of the database user or role. An index on <b>username</b> allows only unique values. The <b>username</b> value can be the login name of a user or the name of a role.
<b>usertype</b>	CHAR(1)	Code specifying the highest database-level privilege held by <b>username</b> , where <b>username</b> 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) 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.

**Table 39. SYSUSERS table column descriptions**

(continued)

Column	Type	Explanation
<b>prior</b>	SMALL	Reserved for future use.
<b>ity</b>	INT	
<b>pass</b>	CHAR(	Reserved for future use.
<b>word</b>	16)	
<b>defr</b>	VARCH	Name of the default role.
<b>ole</b>	AR(32)	

## 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
<b>tabid</b>	INTEGER	Identifying code of the view
<b>seqno</b>	SMALLINT	Line number of the SELECT statement
<b>viewtext</b>	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
<b>targetid</b>	INTEGER	Identifying code of the <i>target table</i> (the base table on which the violations table and the diagnostic table are defined)
<b>viotid</b>	INTEGER	Identifying code of the violations table
<b>diatid</b>	INTEGER	Identifying code of the diagnostics table

Column	Type	Explanation
<b>maxrows</b>	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 **sysxdatasources** system catalog table stores XA data sources.

The **sysxdatasources** table has the following columns.

Column	Type	Explanation
<b>xa_datasrc_owner</b>	CHAR(32)	The user ID of the XA data source owner
<b>xa_datasrc_name</b>	VARCHAR(128)	The name of the XA data source
<b>xa_datasrc_rmid</b>	SERIAL	Unique RMID of the XA data source
<b>xa_source_typeid</b>	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
<b>xa_source_typeid</b>	SERIAL	A unique identifier for the source type
<b>xa_source_owner</b>	CHAR(32)	The user ID of the owner
<b>xa_source_name</b>	VARCHAR(128)	The name of the source type
<b>xa_flags</b>	INTEGER	
<b>xa_version</b>	INTEGER	
<b>xa_open</b>	INTEGER	UDR ID of xa_open_entry
<b>xa_close</b>	INTEGER	UDR ID of xa_close_entry

Column	Type	Explanation
<b>xa_start</b>	INTEGER	UDR ID of xa_start entry
<b>xa_end</b>	INTEGER	UDR ID of xa_end_entry
<b>xa_rollback</b>	INTEGER	UDR ID of xa_rollback_entry
<b>xa_prepare</b>	INTEGER	UDR ID of xa_prepare_entry
<b>xa_commit</b>	INTEGER	UDR ID of xa_commit_entry
<b>xa_recover</b>	INTEGER	UDR ID of xa_recover_entry
<b>xa_forget</b>	INTEGER	UDR ID of xa_forget_entry
<b>xa_complete</b>	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
<b>extended_id</b>	INTEGER	Code uniquely identifying the extended data types
<b>seqno</b>	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.
<b>description</b>	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
<b>grantor</b>	VARCHAR(32)	Name of grantor of privilege
<b>grantee</b>	VARCHAR(32)	Name of grantee of privilege
<b>type</b>	INTEGER	Code identifying the UDT
<b>auth</b>	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 40. SYSXTDTYPES table column descriptions**

Column	Type	Explanation
<b>extended_id</b>	SERIAL	Unique identifying code for extended data type
<b>domain</b>	CHAR(1)	Code for the domain of the UDT
<b>mode</b>	CHAR(1)	Code classifying the UDT: <ul style="list-style-type: none"> <li>• B = Base (opaque) type</li> <li>• C = Collection type or unnamed ROW type</li> <li>• D = Distinct type</li> <li>• R = Named ROW type</li> <li>• S = Reserved for internal use</li> <li>• T = Reserved for internal use</li> <li>• ' ' (blank) = Built-in type</li> </ul>
<b>owner</b>	VARCHAR(32)	Name of the owner of the UDT
<b>name</b>	VARCHAR(128)	Name of the UDT
<b>type</b>	SMALLINT	Code classifying the UDT
<b>source</b>	INTEGER	The <b>sysxtdtypes</b> reference (for distinct types only)

**Table 40. SYSXTDTYPES table column descriptions**

(continued)

Column	Type	Explanation
		Zero (0) indicates that a distinct UDT was created from a built-in data type.
<b>maxlen</b>	INTEGER	The maximum length for variable-length data types  Zero indicates a fixed-length UDT.
<b>length</b>	INTEGER	The length in bytes for fixed-length data types  Zero indicates a variable-length UDT.
<b>byvalue</b>	CHAR(1)	'T' = UDT is passed by value  'F' = UDT is not passed by value
<b>cannothash</b>	CHAR(1)	'T' = UDT is hashable by default hash function  'F' = UDT is not hashable by default function
<b>align</b>	SMALLINT	Alignment ( = 1, 2, 4, or 8) for this UDT
<b>locator</b>	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 23](#), 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 DB-Access 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
<b>table_schema</b>	VARCHAR(32)	Name of owner of table
<b>table_name</b>	VARCHAR(128)	Name of table or view
<b>table_type</b>	VARCHAR(128)	BASE TABLE for table or VIEW for view
<b>remarks</b>	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 41. Description of the columns Information Schema View**

Column	Data Type	Explanation
<b>table_schema</b>	VARCHAR(128)	Name of owner of table
<b>table_name</b>	VARCHAR(128)	Name of table or view
<b>column_name</b>	VARCHAR(128)	Name of the column in the table or view
<b>ordinal_position</b>	INTEGER	Position of the column within its table  The <b>ordinal_position</b> value is a sequential number that starts at 1 for the first column. This is the HCL Informix® extension to XPG4.
<b>data_type</b>	VARCHAR(254)	Name of the data type of the column, such as CHARACTER or DECIMAL

Table 41. Description of the columns Information Schema View (continued)

Column	Data Type	Explanation
<b>char_max_length</b>	INTEGER	Maximum length (in bytes) for character data types; NULL otherwise
<b>numeric_precision</b>	INTEGER	Uses one of the following values: <ul style="list-style-type: none"> <li>• Total number of digits for exact numeric data types (DECIMAL, INTEGER, MONEY, SMALLINT)</li> <li>• Number of digits of mantissa precision (machine-dependent) for approximate data types (FLOAT, SMALLFLOAT)</li> <li>• NULL for all other data types.</li> </ul>
<b>numeric_prec_radix</b>	INTEGER	Uses one of the following values: <ul style="list-style-type: none"> <li>• 2 = Approximate data types (FLOAT and SMALLFLOAT)</li> <li>• 10 = Exact numeric data types (DECIMAL, INTEGER, MONEY, and SMALLINT)</li> <li>• NULL for all other data types</li> </ul>
<b>numeric_scale</b>	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
<b>datetime_precision</b>	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.
<b>is_nullable</b>	VARCHAR(3)	Indicates whether a column allows NULL values; either YES or NO
<b>remarks</b>	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
<b>source</b>	VARCHAR(254)	Organization defining this SQL version
<b>source_year</b>	VARCHAR(254)	Year the source document was approved
<b>conformance</b>	VARCHAR(254)	Standard to which the server conforms
<b>integrity</b>	VARCHAR(254)	Indication of whether this is an integrity enhancement feature; either <code>YES</code> or <code>NO</code>
<b>implementation</b>	VARCHAR(254)	Identification of the SQL product of the vendor
<b>binding_style</b>	VARCHAR(254)	Direct, module, or other binding style
<b>programming_lang</b>	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
<b>server_attribute</b>	VARCHAR(254)	An attribute of the database server
<b>attribute_value</b>	VARCHAR(254)	Value of the <code>server_attribute</code> 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
<b>identifier_length</b>	Maximum number of bytes for a user-defined identifier
<b>row_length</b>	Maximum number of bytes in a row
<b>userid_length</b>	Maximum number of bytes in a user name
<b>txn_isolation</b>	Initial transaction isolation level for the database server:  <code>Read Uncommitted</code> (= Default isolation level for databases with no transaction logging; also called <code>Dirty Read</code> )  <code>Read Committed</code> (= 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 42. Data types supported in all operations**

Data type	Explanation
<a href="#">BIGINT data type on page 86</a>	Stores 8-byte integer values from $-(2^{63}-1)$ to $2^{63}-1$
<a href="#">BIGSERIAL data type on page 87</a>	Stores sequential, 8-byte integers from 1 to $2^{63}-1$
<a href="#">BSON and JSON built-in opaque data types on page</a>	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.
<a href="#">BYTE data type on page 89</a>	Stores any kind of binary data, up to $2^{31}$ bytes in length
<a href="#">CHAR(n) data type on page 90</a>	Stores character strings; collation is in code-set order
<a href="#">CHARACTER(n) data type on page 91</a>	Is a synonym for CHAR
<a href="#">CHARACTER VARYING(m,r) data type on page 91</a>	Stores character strings of varying length (ANSI-compliant); collation is in code-set order
<a href="#">DATE data type on page 92</a>	Stores calendar dates
<a href="#">DATETIME data type on page 93</a>	Stores calendar date combined with time of day
<a href="#">DEC data type on page 96</a>	Is a synonym for DECIMAL
<a href="#">DECIMAL on page 96</a>	Stores floating-point numbers with definable precision; if database is ANSI-compliant, the scale is zero
<a href="#">DECIMAL (p,s) Fixed Point on page 97</a>	Stores fixed-point numbers of defined scale and precision
<a href="#">DOUBLE PRECISION data types on page 99</a>	Synonym for FLOAT
<a href="#">FLOAT(n) on page 99</a>	Stores double-precision floating-point numbers corresponding to the <b>double</b> data type in C
<a href="#">INT data type on page 100</a>	Is a synonym for INTEGER
<a href="#">INT8 on page 100</a>	Stores 8-byte integer values from $-(2^{63}-1)$ to $2^{63}-1$

**Table 42. Data types supported in all operations (continued)**

Data type	Explanation
<a href="#">INTEGER data type on page 100</a>	Stores whole numbers from -2,147,483,647 to +2,147,483,647
<a href="#">INTERVAL data type on page 101</a>	Stores a span of time (or level of effort) in units of <i>years</i> and <i>months</i> .
<a href="#">INTERVAL data type on page 101</a>	Stores a span of time in a contiguous set of units of <i>days</i> , <i>hours</i> , <i>minutes</i> , <i>seconds</i> , and <i>fractions of a second</i>
<a href="#">MONEY(p,s) data type on page 105</a>	Stores currency amounts
<a href="#">NCHAR(n) data type on page 107</a>	Same as CHAR, but can support localized collation
<a href="#">NUMERIC(p,s) data type on page 107</a>	Synonym for DECIMAL(p,s)
<a href="#">NVARCHAR(m,r) data type on page 107</a>	Same as VARCHAR, but can support localized collation
<a href="#">REAL data type on page 108</a>	Is a synonym for SMALLFLOAT
<a href="#">SERIAL(n) data type on page 111</a>	Stores sequential integers (> 0) in positive range of INT
<a href="#">SERIAL8(n) data type on page 112</a>	Stores sequential integers (> 0) in positive range of INT8
<a href="#">SMALLFLOAT on page 114</a>	Stores single-precision floating-point numbers corresponding to the <b>float</b> data type of the C language
<a href="#">SMALLINT data type on page 115</a>	Stores whole numbers from -32,767 to +32,767
<a href="#">TEXT data type on page 115</a>	Stores any kind of text data, up to 2 <sup>31</sup> bytes in length
<a href="#">VARCHAR(m,r) data type on page 117</a>	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 43. Data types supported in a local database**

Data type	Explanation
<a href="#">BLOB data type on page 87</a>	Stores binary data in random-access chunks
<a href="#">The binary18 data type on page</a>	Stores 18 byte binary-encoded strings
<a href="#">The binaryvar data type on page</a>	Stores binary-encoded strings with a maximum length of 255 bytes
<a href="#">BOOLEAN data type on page 88</a>	Stores Boolean values true and false
<a href="#">CLOB data type on page 91</a>	Stores text data in random-access chunks
<a href="#">DISTINCT data types on page 98</a>	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
<a href="#">Calendar data type on page</a>	Stores a calendar for a TimeSeries data type
<a href="#">CalendarPattern data type on page</a>	Stores the structure of the calendar pattern for a Calendar data type
<a href="#">IDSSECURITYLABEL data type on page 100</a>	Stores LBAC security label objects.
<a href="#">LIST(e) data type on page 103</a>	Stores a sequentially ordered collection of elements, all of the same data type, e; allows duplicate values
<a href="#">The lld_locator data type on page</a>	Stores a large object identifier
<a href="#">The lld_job data type on page</a>	Stores the location of a smart large object and specifies whether the object contains binary or character data
<a href="#">LVARCHAR(m) data type on page 104</a>	Stores variable-length strings of up to 32,739 bytes
<a href="#">MULTISET(e) data type on page 106</a>	Stores a non-ordered collection of values, with elements all of the same data type, e; allows duplicate values.
<a href="#">The node data type for querying hierarchical data on page</a>	Stores a combination of integers and decimal points that represents hierarchical relationships, of variable length up to 256 characters
<a href="#">OPAQUE data types on page 108</a>	Stores a user-defined data type whose internal structure is inaccessible to the database server



**Table 43. Data types supported in a local database (continued)**

Data type	Explanation
<a href="#">ROW data type, Named on page 108</a>	Stores a named ROW type
<a href="#">ROW data type, Unnamed on page 110</a>	Stores an unnamed ROW type
<a href="#">SET(e) data type on page 113</a>	Stores a non-ordered collection of elements, all of the same data type, e; does not allow duplicate values
<a href="#">ST_LineString data type on page</a>	Stores a one-dimensional object as a sequence of points defining a linear interpolated path
<a href="#">ST_MultiLineString data type on page</a>	Stores a collection of ST_LineString data types
<a href="#">ST_MultiPoint data type on page</a>	Stores a collection of ST_Point data types
<a href="#">ST_MultiPolygon data type on page</a>	Stores a collection of ST_Polygon data types
<a href="#">ST_Point data type on page</a>	Stores a zero-dimensional geometry that occupies a single location in coordinate space
<a href="#">ST_Polygon data type on page</a>	Stores a two-dimensional surface stored as a sequence of points defining its exterior bounding ring and 0 or more interior rings
<a href="#">TimeSeries data type on page</a>	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 reference

[Built-In Data Types on page 118](#)

### Related information

[Extended Data Types on page 129](#)

## 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 111](#)
- Using the SERIAL8 data type with the INT8 or BIGINT data type, see [Using SERIAL8 and BIGSERIAL with INT8 or BIGINT on page 87](#)

## 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 ( $4 \times 2^{40}$  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 91](#)) for random access to text data. For general information about BLOB and CLOB data types, see [Smart large objects on page 122](#).

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 (IBM® Informix® ESQL/C)

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 (DB-Access)
- From BYTE host variables (IBM® Informix® ESQL/C )

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 DB-Access 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.

## CHAR(n) data type

The CHAR data type stores any string of letters, numbers, and symbols. It can store single-byte and multibyte characters, based on the database locale.

A CHAR(*n*) column has a length of *n* bytes, where  $1 \leq n \leq 32,767$ . If you do not specify *n*, CHAR(1) is the default length. Character columns typically store alphanumeric strings, such as names, addresses, phone numbers, and so on. When a value is retrieved or stored as CHAR(*n*), exactly *n* bytes of data are transferred. If the string is shorter than *n* bytes, the string is extended with blank spaces up to the declared length. If the data value is longer than *n* bytes, a data string of length *n* that has been truncated from the right is inserted or retrieved, without the database server raising an exception.

This does not create partial characters in multibyte locales. In right-to-left locales, such as Arabic, Hebrew, or Farsi, the truncation is from the left.

Size specifications in CHAR 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 120](#).

For more information about East Asian locales that support multibyte code sets, see [Multibyte Characters with VARCHAR on page 118](#).

## Treating CHAR Values as Numeric Values

If you plan to perform calculations on numbers stored in a column, you should assign a number data type to that column. Although you can store numbers in CHAR columns, you might not be able to use them in some arithmetic operations. For example, if you insert a sum into a CHAR column, you might experience overflow problems if the CHAR column is too small to hold the value. In this case, the insert fails. Numbers that have leading zeros (such as some zip codes) have the zeros stripped if they are stored as number types INTEGER or SMALLINT. Instead, store these numbers in CHAR columns.

## Sorting and Relational Comparisons

In general, the collating order for sorting CHAR values is the order of characters in the code set. (An exception is the MATCHES operator with ranges; see [Collating VARCHAR Values on page 118](#).) For more information about collation order, see the *HCL® Informix® GLS User's Guide*.

For multibyte locales, the database supports any multibyte characters in the code set. When storing multibyte characters in a CHAR data type, make sure to calculate the number of bytes needed. For more information about multibyte characters and locales, see the *HCL® Informix® GLS User's Guide*.

CHAR values are compared to other CHAR values by padding the shorter value on the right with blank spaces until the values have equal length, and then comparing the two values, using the code-set order for collation.

## Nonprintable Characters with CHAR

A CHAR value can include tab, newline, whitespace, and nonprintable characters. You must, however, use an application to insert nonprintable characters into host variables and the host variables into your database. After passing nonprintable characters to the database server, you can store or retrieve them. After you select nonprintable characters, fetch them into host variables and display them with your own display mechanism.

An important exception is the first value in the ASCII code set is used as the end-of-data terminator symbol in columns of the CHAR data type. For this reason, any subsequent characters in the same string cannot be retrieved from a CHAR column, because the database server reads only the characters (if any) that precede this null terminator. For example, you cannot use the following 7-byte string as a CHAR data type value with a length of 7 bytes:

```
abc\0def
```

If you try to display nonprintable characters with DB-Access your screen returns inconsistent results. (Which characters are nonprintable is locale-dependent. For more information see the discussion of code-set conversion between the client and the database server in the *HCL® Informix® GLS User's Guide*.)

## 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 117](#).

## 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 ( $4 \times 2^{40}$  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 87](#)) for random access to binary data. For general information about the CLOB and BLOB data types, see [Smart large objects on page 122](#).

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 157](#).

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*.

## DATETIME data type

The DATETIME data type stores an instant in time expressed as a calendar date and time of day.

You select how precisely a DATETIME value is stored; its precision can range from a year to a fraction of a second.

DATETIME stores a data value as a contiguous series of fields that represents each time unit (*year, month, day*, and so forth) in the data type declaration.

Field qualifiers to specify a DATETIME data type have this format:

```
DATETIME largest_qualifier TO smallest_qualifier
```

This resembles an INTERVAL field qualifier, but DATETIME represents a point in time, rather than (like INTERVAL) a span of time. These differences exist between DATETIME and INTERVAL qualifiers:

- The DATETIME keyword replaces the INTERVAL keyword.
- DATETIME field qualifiers cannot specify a nondefault precision for the *largest\_qualifier* time unit.
- Field qualifiers of a DATETIME data type can include YEAR, MONTH, and smaller time units, but an INTERVAL data type that includes the DAY field qualifier (or smaller time units) cannot also include the YEAR or MONTH field qualifiers.

The *largest\_qualifier* and *smallest\_qualifier* of a DATETIME data type can be any of the fields that the following table lists, provided that *smallest\_qualifier* does not specify a larger time unit than *largest\_qualifier*. (The largest and smallest time units can be the same; for example, DATETIME YEAR TO YEAR.)

**Table 44. DATETIME field qualifiers**

Qualifier field	Valid entries
YEAR	A year numbered from 1 to 9,999 (A.D.)

**Table 44. DATETIME field qualifiers (continued)**

Qualifier field	Valid entries
MONTH	A month numbered from 1 to 12
DAY	A day numbered from 1 to 31, as appropriate to the month
HOUR	An hour numbered from 0 (midnight) to 23
MINUTE	A minute numbered from 0 to 59
SECOND	A second numbered from 0 - 59
FRACTION	A decimal fraction-of-a-second with up to 5 digits of scale. The default scale is 3 digits (a thousandth of a second). For <i>smallest_qualifier</i> to specify another scale, write FRACTION( <i>n</i> ), where <i>n</i> is the number of digits from 1 - 5.

The declaration of a DATETIME column need not include the full YEAR to FRACTION range of time units. It can include any contiguous subset of these time units, or even only a single time unit.

For example, you can enter a MONTH TO HOUR value in a column declared as YEAR TO MINUTE, if each entered value contains information for a contiguous series of time units. You cannot, however, enter a value for only the MONTH and HOUR; the entry must also include a value for DAY.

If you use the DB-Access TABLE menu, and you do not specify the DATETIME qualifiers, a default DATETIME qualifier, YEAR TO YEAR, is assigned.

A valid DATETIME literal must include the DATETIME keyword, the values to be entered, and the field qualifiers. You must include these qualifiers because, as noted earlier, the value that you enter can contain fewer fields than were declared for that column. Acceptable qualifiers for the first and last fields are identical to the list of valid DATETIME fields that are listed in the table [Table 44: DATETIME field qualifiers on page 93](#).

Write values for the field qualifiers as integers and separate them with delimiters. The following table lists the delimiters that are used with DATETIME values in the default US English locale. (These are a superset of the delimiters that are used in INTERVAL values.)

**Table 45. Delimiters used with DATETIME**

Delimiter	Placement in DATETIME Literal
Hyphen ( - )	Between the YEAR, MONTH, and DAY time-unit values
Blank space ( )	Between the DAY and HOUR time-unit values
Colon ( : )	Between the HOUR, MINUTE, and SECOND time-unit values
Decimal point ( . )	Between the SECOND and FRACTION time-unit values

The following illustration shows a DATETIME YEAR TO FRACTION(3) value with delimiters.

Figure 1. Example DATETIME Value with Delimiters

2003-09-23 12:42:06.001

↑ year      ↑ Month      ↑ Day      ↑ Hour      ↑ Minute      ↑ Second      ↑ Fraction

When you enter a value with fewer time-unit fields than in the column, the value that you enter is expanded automatically to fill all the declared time-unit fields. If you leave out any more significant fields, that is, time units larger than any that you include, those fields are filled automatically with the current values for those time units from the system clock calendar. If you leave out any less-significant fields, those fields are filled with zeros (or with `1` for MONTH and DAY) in your entry.

You can also enter DATETIME values as character strings. The character string must include information for each field defined in the DATETIME column. The INSERT statement in the following example shows a DATETIME value entered as a character string:

```
INSERT INTO cust_calls (customer_num, call_dtime, user_id,
    call_code, call_descr)
VALUES (101, '2001-01-14 08:45', 'maryj', 'D',
    'Order late - placed 6/1/00');
```

If **call\_dtime** is declared as DATETIME YEAR TO MINUTE, the character string must include values for the *year*, *month*, *day*, *hour*, and *minute* fields.

If the character string does not contain information for all the declared fields (or if it adds additional fields), then the database server returns an error.

All fields of a DATETIME column are two-digit numbers except for the *year* and *fraction* fields. The *year* field is stored as four digits. When you enter a two-digit value in the year field, how the abbreviated year is expanded to four digits depends on the setting of the **DBCENTURY** environment variable.

For example, if you enter `02` as the year value, whether the year is interpreted as `1902`, `2002`, or `2102` depends on the setting of **DBCENTURY** and on the value of the system clock calendar at execution time. If you do not set **DBCENTURY**, the leading digits of the current year are appended by default.

The *fraction* field requires  $n$  digits where  $1 \leq n \leq 5$ , rounded up to an even number. You can use the following formula (rounded up to a whole number of bytes) to calculate the number of bytes that a DATETIME value requires:

$$(total\ number\ of\ digits\ for\ all\ fields) / 2 + 1$$

For example, a YEAR TO DAY qualifier requires a total of eight digits (four for *year*, two for *month*, and two for *day*). According to the formula, this data value requires 5, or  $(8/2) + 1$ , bytes of storage.

The USEOSTIME configuration parameter can affect the subsecond granularity when the database server obtains the current time from the operating system in SQL statements. For details, see the *HCL® Informix® Administrator's Reference*.

With an ESQL API, the **DBTIME** environment variable affects DATETIME formatting. Nondefault locales and settings of the **GL\_DATE** and **DBDATE** environment variables also affect the display of datetime data. They do not, however, affect the internal storage format of a DATETIME column.

If you specify a locale other than U.S. English, the locale defines the culture-specific display formats for DATETIME values. To change the default display format, change the setting of the **GL\_DATETIME** environment variable. When a database with a nondefault locale uses a nondefault **GL\_DATETIME** setting, the **USE\_DTEENV** environment variable must be set to `1` before the database server can correctly process localized DATETIME values in the following operations:

- using the LOAD or UNLOAD feature of DB-Access
- using the dbexport or dbimport migration utilities
- using DML statements of SQL on database tables or on objects that the CREATE EXTERNAL TABLE statement defined.

For more information about locales and GLS environment variables that can specify end-user DATETIME formats, see the *HCL® Informix® GLS User's Guide*.

---

#### Related reference

[INTERVAL data type on page 101](#)

[DBCENTURY environment variable on page 154](#)

[DBTIME environment variable on page 167](#)

#### Related information

[Manipulating DATE with DATETIME and INTERVAL Values on page 126](#)

[Manipulating DATETIME Values on page 124](#)

[The mi\\_datetime\\_compare\(\) function on page](#)

## 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

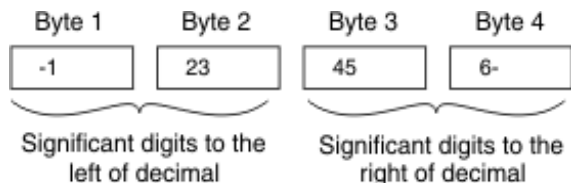
The database server uses one byte of disk storage to store two digits of a decimal number, plus an additional byte to store the exponent and sign, with the first byte representing a sign bit and a 7-bit exponent in excess-65 format. The rest of the bytes express the mantissa as base-100 digits. The significant 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 2: Schematic that illustrates the storage of digits in a decimal \(p,s\) value on page 98](#) 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 2. 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 **sysxdtypes** 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 138](#). 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 120](#).

## DOUBLE PRECISION data types

The DOUBLE PRECISION keywords are a synonym for the FLOAT keyword.

### Related reference

[FLOAT\(n\) on page 99](#)

## 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 reference**[DOUBLE PRECISION data types on page 99](#)

## 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.

## INTERVAL data type

The INTERVAL data type stores a value that represents a span of time. INTERVAL types are divided into two classes: *year-month intervals* and *day-time intervals*.

A year-month interval can represent a span of years and months, and a day-time interval can represent a span of days, hours, minutes, seconds, and fractions of a second.

An INTERVAL value is always composed of one value or a series of values that represents time units. Within a data-definition statement such as CREATE TABLE or ALTER TABLE that defines the precision of an INTERVAL data type, the qualifiers must have the following format:

```
INTERVAL largest_qualifier(n) TO smallest_qualifier
```

Here the *largest\_qualifier* and *smallest\_qualifier* keywords are taken from one of the two INTERVAL classes, as shown in the table [Table 46: Interval Classes on page 101](#).

If SECOND (or a larger time unit) is the *largest\_qualifier*, the declaration of an INTERVAL data type can optionally specify *n*, the precision of the largest time unit (for *n* ranging from 1 to 9); this is not a feature of DATETIME data types.

If *smallest\_qualifier* is FRACTION, you can also specify a scale in the range from 1 to 5. For FRACTION TO FRACTION qualifiers, the upper limit of *n* is 5, rather than 9. There are two incommensurable classes of INTERVAL data types:

- Those with a *smallest\_qualifier* larger than DAY
- Those with a *largest\_qualifier* smaller than MONTH

**Table 46. Interval Classes**

Interval Class	Time Units	Valid Entry
YEAR-MONTH INTERVAL	YEAR	A number of years
YEAR-MONTH INTERVAL	MONTH	A number of months
DAY-TIME INTERVAL	DAY	A number of days
DAY-TIME INTERVAL	HOURL	A number of hours
DAY-TIME INTERVAL	MINUTE	A number of minutes
DAY-TIME INTERVAL	SECOND	A number of seconds
DAY-TIME INTERVAL	FRACTION	A decimal fraction of a second, with up to 5 digits. The default scale is 3 digits (thousandth of a second). To specify a non-default scale, write FRACTION( <i>n</i> ), where $1 \leq n \leq 5$ .

As with DATETIME data types, you can define an INTERVAL to include only the subset of time units that you need. But because the construct of "month" (as used in calendar dates) is not a time unit that has a fixed number of days, a single INTERVAL value cannot combine months and days; arithmetic that involves operands of the two different INTERVAL classes is not supported.

A value entered into an INTERVAL column need not include the full range of time units that were specified in the data-type declaration of the column. For example, you can enter a value of HOUR TO SECOND precision into a column defined as DAY TO SECOND. A value must always consist, however, of contiguous time units. In the previous example, you cannot enter only the HOUR and SECOND values; you must also include MINUTE values.

A valid INTERVAL literal contains the INTERVAL keyword, the values to be entered, and the field qualifiers. (See the discussion of literal intervals in the *HCL® Informix® Guide to SQL: Syntax*.) When a value contains only one field, the largest and smallest fields are the same.

When you enter a value in an INTERVAL column, you must specify the largest and smallest fields in the value, just as you do for DATETIME values. In addition, you can optionally specify the precision of the first field (and the scale of the last field if it is a FRACTION). If the largest and smallest field qualifiers are both FRACTION, you can specify only the scale in the last field.

Acceptable qualifiers for the largest and smallest fields are identical to the list of INTERVAL fields that the tab;e [Table 46: Interval Classes on page 101](#) displays.

If you use the DB-Access **TABLE** menu, but you specify no INTERVAL field qualifiers, then a default INTERVAL qualifier, YEAR TO YEAR, is assigned.

The *largest\_qualifier* in an INTERVAL value can be up to nine digits (except for FRACTION, which cannot be more than five digits), but if the value that you want to enter is greater than the default number of digits allowed for that field, you must explicitly identify the number of significant digits in the value that you enter. For example, to define an INTERVAL of DAY TO HOUR that can store up to 999 days, you can specify it the following way:

```
INTERVAL DAY(3) TO HOUR
```

INTERVAL literals use the same delimiters as DATETIME literals (except that MONTH and DAY time units are not valid within the same INTERVAL value). the following table shows the INTERVAL delimiters.

**Table 47. INTERVAL Delimiters**

Delimiter	Placement in an INTERVAL Literal
Hyphen	Between the YEAR and MONTH portions of the value
Blank space	Between the DAY and HOUR portions of the value
Colon	Between the HOUR, MINUTE, and SECOND portions of the value
Decimal point	Between the SECOND and FRACTION portions of the value

You can also enter INTERVAL values as character strings. The character string must include information for the same time units that were specified in the data-type declaration for the column. The INSERT statement in the following example shows an INTERVAL value entered as a character string:

```
INSERT INTO manufact (manu_code, manu_name, lead_time)
VALUES ('BRO', 'Ball-Racquet Originals', '160')
```

Because the **lead\_time** column is defined as INTERVAL DAY(3) TO DAY, this INTERVAL value requires only one field, the span of days required for lead time. If the character string does not contain information for all fields (or adds additional fields), the database server returns an error. For additional information about entering INTERVAL values as character strings, see the *HCL® Informix® Guide to SQL: Syntax*.

By default, all fields of an INTERVAL column are two-digit numbers, except for the year and fraction fields. The year field is stored as four digits. The fraction field requires  $n$  digits where  $1 \leq n \leq 5$ , rounded up to an even number. You can use the following formula (rounded up to a whole number of bytes) to calculate the number of bytes required for an INTERVAL value:

$$(total\ number\ of\ digits\ for\ all\ fields) / 2 + 1$$

For example, INTERVAL YEAR TO MONTH requires six digits (four for *year* and two for *month*), and requires 4, or  $(6/2) + 1$ , bytes of storage.

For information about using INTERVAL as a constant expression, see the description of the INTERVAL Field Qualifier in the *HCL® Informix® Guide to SQL: Syntax*.

#### Related reference

[DATETIME data type on page 93](#)

#### Related information

[Manipulating DATE with DATETIME and INTERVAL Values on page 126](#)

[Manipulating INTERVAL Values on page 128](#)

[The mi\\_interval\\_compare\(\) function on page](#)

## 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 106](#) and [SET\(e\) data type on page 113](#). For complex data types that can store a set of values that includes different SQL data types, see [ROW Data Types on page 132](#).

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 120](#).

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.

Like 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 [DBMONEY environment variable on page 162](#).

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 103](#).)

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 108](#).

## 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 in 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 120](#).

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 118](#) 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 120](#).

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 **sysxtotypes** 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 132](#).

## 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 **sysxtotypes** 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 DB-Access 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:

$$(\text{maximum existing value in SERIAL column}) + 1$$

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 100](#)).

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 DB-Access 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 113](#). 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 87](#)

## 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  $n$  in a SERIAL8 column of a table, give the value  $n$  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 87](#)

## 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 MULTISSET in [MULTISSET\(e\) data type on page 106](#).)

No more than 97 columns of the same table can be declared as SET data types. (The same restriction also applies to MULTISSET 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 103](#).) 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 (DB-Access)
- 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 on page 91](#).

## Unnamed ROW

See [ROW data type, Unnamed on page 110](#).

## 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 182](#) 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 on page 91](#).

## 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 120](#).

## 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	<a href="#">BSON and JSON built-in opaque data types on page</a>
Numeric	BIGINT, BIGSERIAL, DECIMAL, FLOAT, INT8, INTEGER, MONEY, SERIAL, SERIAL8, SMALLFLOAT, SMALLINT
Time	DATE, DATETIME, INTERVAL

**Related reference**

[Summary of data types on page 81](#)

**Related information**

[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 48. Attributes of built-in character data types**

	Size (in bytes)	Default	Reserved	Collation	Length
<b>CHAR(n)</b>	1 to 32,767	1 byte	None	Code set	Fixed
<b>NCHAR(n)</b>	1 to 32,767	1 byte	None	Localized	Fixed
<b>VARCHAR(m, r)</b>	1 to 255	0 for r	0 to 255 bytes	Code set	Variable
<b>NVARCHAR(m, r)</b>	1 to 255	0 for r	0 to 255 bytes	Localized	Variable
<b>LVARCHAR(m)</b>	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 MULTISSET.

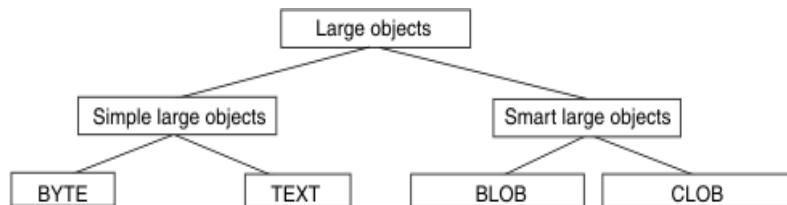
In a database that is insensitive to the letter case of NCHAR or NVARCHAR values, string manipulation operations might produce unexpected results, if they implicitly cast CHAR, LVARCHAR, or VARCHAR operands or arguments to NCHAR or NVARCHAR data types. In some contexts, the operation can return a duplicate 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 3: Large-Object Data Types on page 122](#) shows the large-object data types.

Figure 3. 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 89](#).

### TEXT

Stores text data. For more detailed information about this data type, see the description on page [TEXT data type on page 115](#).

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 132](#).) 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 87](#).

**CLOB**

Stores text data. For more information about this data type, see [CLOB data type on page 91](#).

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 IBM® Informix® ESQL/C programs, use the smart-large-object API. (For more information, see the *HCL® Informix® Enterprise Replication Guide*.)
- From within a DataBlade® 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 49: Arithmetic Operations on DATE, DATETIME, and INTERVAL Values on page 124](#) 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 49. 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
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 reference

[DATETIME data type on page 93](#)

#### Related information

[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 50: Results of Expressions That Manipulate DATE with DATETIME or INTERVAL Values on page 126](#) shows.

**Table 50. 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 50: Results of Expressions That Manipulate DATE with DATETIME or INTERVAL Values on page 126](#) 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 157](#) and [DBTIME environment variable on page 167](#). 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 reference

[DATETIME data type on page 93](#)

[INTERVAL data type on page 101](#)

## 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 reference

[INTERVAL data type on page 101](#)

#### Related information

[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 reference

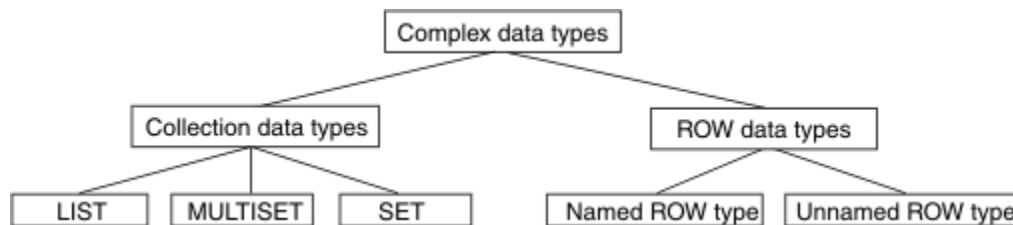
[Summary of data types on page 81](#)

## Complex data types

A *complex data type* can store one or more values of other built-in or extended data types.

Figure 4: [Complex Data Types of HCL Informix on page 129](#) shows the complex types that HCL Informix® supports.

Figure 4. Complex Data Types of HCL Informix®



The following table summarizes the structure of the complex data types.

**Table 51. 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.

**Table 51. Collection types are complex data types that are made up of elements, each of which is of the same data type.**  
(continued)

Collection types	Description
SET	A group of elements, each of which is unique. The order of the elements is ignored.

**Table 52. 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 IBM® Informix® ESQL/C 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 108](#) and [ROW data type, Unnamed on page 110](#).

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 MULTISSET). 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 98](#). 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 10](#), have columns of built-in opaque data types. For information on how the system catalog encodes columns of built-in opaque data types, see [SYSCOLUMNS on page 23](#).

## 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 sysxtotypes 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 108](#). 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 20](#).

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 53. 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 54. Extended data type combinations**

Target Type	Opaque Type	Distinct Type	Named ROW Type	Unnamed ROW Type	Collection Type	Built-in Type
<b>Opaque Type</b>	Explicit or implicit	Explicit	Explicit	Not Valid	Not Valid	Explicit or implicit <sup>3</sup>
<b>Distinct Type</b>	Explicit <sup>3</sup>	Explicit	Explicit	Not Valid	Not Valid	Explicit or implicit
<b>Named ROW Type</b>	Explicit <sup>3</sup>	Explicit	Explicit <sup>3</sup>	Explicit <sup>1</sup>	Not Valid	Not Valid
<b>Unnamed ROW Type</b>	Not Valid	Not Valid	Explicit <sup>1</sup>	Implicit <sup>1</sup>	Not Valid	Not Valid
<b>Collection Type</b>	Not Valid	Not Valid	Not Valid	Not Valid	Explicit <sup>2</sup>	Not Valid
<b>Built-in Type</b>	Explicit or implicit <sup>3</sup>	Explicit or implicit	Not Valid	Not Valid	Not Valid	System defined (implicit)

<sup>1</sup> Applies when two ROW types are structurally equivalent or casts exist to handle data conversions where corresponding field types are not the same.

<sup>2</sup> Applies when a cast exists to convert between the element types of the respective collection types.

<sup>3</sup> 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 that HCL 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
. ( <i>membership</i> ) [] ( <i>substring</i> )	<b>customer.phone</b> [1, 3]
UNITS	<b>x</b> UNITS DAY
+ - ( <i>unary</i> )	- <b>y</b>
:: ( <i>cast</i> )	NULL::TEXT
* /	<b>x / y</b>
+ - ( <i>binary</i> )	<b>x - y</b>
( <i>concatenation</i> )	<b>customer.fname</b>    <b>customer.lname</b>
ANY ALL SOME	<b>orders.ship_date</b> > SOME (SELECT <b>paid_date</b> FROM <b>orders</b> )
NOT	NOT <b>y</b>
< <= = > >= != <>	<b>x &gt;= y</b>
IN BETWEEN ... AND LIKE MATCHES	<b>customer.fname</b> MATCHES <b>y</b>
AND	<b>x</b> AND <b>y</b>
OR	<b>x</b> OR <b>y</b>

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 #unique\_2015\_Connect\_42\_sii-03-39744.

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 202](#). For more information about these routines and on the SET ENVIRONMENT statement, see the *HCL® Informix® Guide to SQL: Syntax*.

In IBM® Informix® ESQL/C, 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 174](#), 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 144](#), 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	<code>setenv ABCD value</code>
Bourne	<code>ABCD=value</code> <code>export ABCD</code>
Korn	<code>ABCD=value</code> <code>export ABCD</code>
Korn	<code>export ABCD=value</code>

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.

```
setenvABCDvalue
```

## Unsetting environment variables

### About this task

To unset an environment variable, enter the following command.

Shell	Command
C	<code>unsetenv ABCD</code>
Bourne or Korn	<code>unset ABCD</code>

## 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 174](#).

## 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 146](#).

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 IBM® Informix® ESQL/C 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 Informix® ESQL/C, 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™.

```
set.ABCD=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 146](#).

---

### 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](#)

## 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 55. Lettercase of implicit, unquoted, and quoted owner names, with and without ANSIOWNER**

Owner Format	Specification	ANSIOWNER = 1	ANSIOWNER Not Set
<b>Implicit:</b>	oblong	owen.oblong	OWEN.oblong
<b>Unquoted:</b>	owen.oblong	owen.oblong	OWEN.oblong
<b>Quoted:</b>	'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 IBM® Informix® ESQL/C source files in your programming environment.

```
setenvCPFIRST { TRUE | FALSE }
```

When you compile Informix® ESQL/C programs with **CPFIRST** not set, the Informix® ESQL/C 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 Informix® ESQL/C program source file in the following order:

1. Run the C preprocessor
2. Run the Informix® ESQL/C 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 Informix® ESQL/C preprocessor on all Informix® ESQL/C 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.

```
setenvCMCONFIGpath/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

IBM 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
↓σνιπδ

onstat -g env 201

IBM 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.

Setting	Algorithm
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 reference

[DATETIME data type on page 93](#)

## 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, such as Hebrew or Islamic calendars. 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.

```
setenv DBDATE { 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.

Some East Asian locales support additional syntax for era-based dates.

**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:

<b>DBDATE Setting</b>	<b>Representation of January 8, 2011:</b>		<b>DBDATE Setting</b>	<b>Representation of January 8, 2011:</b>
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 *IBM® Informix® ESQL/C* 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 [DBTIME environment variable on page 167](#) 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 156](#) 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.

```
setenv DBDELIMITER '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 DB-Access utility displays the floating-point values of data types FLOAT, SMALLFLOAT, and DECIMAL(*p*) within a 14-character buffer. By default, DB-Access 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 DB-Access 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.

```
setenvDBFLTMASKScale
```

**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, DB-Access rounds the value to the specified number of digits. If the floating-point value contains fewer digits to the right of the decimal, DB-Access pads the value with zeros. If you set **DBFLTMASK** to a value greater than can fit into the 14-character buffer, however, DB-Access 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*.

## DBMONEY environment variable

Use the **DBMONEY** environment variable to specify the display format of values in columns of smallfloat, FLOAT, DECIMAL, or MONEY data types, and of complex data types derived from any of these data types.

```
setenvDBMONEY {'$' / front / 'front '}{_ | . }[{'back' / 'back'}]
```

### \$

is a currency symbol that precedes MONEY values in the default locale if no other *front* symbol is specified, or if **DBMONEY** is not set.

### , or .

is a comma or period (the default) that separates the integral part from the fractional part of the FLOAT, DECIMAL, or MONEY value. Whichever symbol you do not specify becomes the thousands separator.

### back

is a currency symbol that follows the MONEY value.

### front

is a currency symbol that precedes the MONEY value.

The *back* symbol can be up to seven characters and can contain any character that the locale supports, except a digit, a comma ( , ), or a period ( . ) symbol. The *front* symbol can be up to seven characters and can contain any character that the locale supports except a digit, a comma ( , ), or a period ( . ) symbol. If you specify any character that is not a letter of the alphabet for *front* or *back*, you must enclose the *front* or *back* setting between single quotation ( ' ) marks.

When you display MONEY values, HCL Informix® products use the **DBMONEY** setting to format the output. **DBMONEY** has no effect, however, on the internal format of data values that are stored in columns of the database.

If you do not set **DBMONEY**, then MONEY values for the default locale, U.S. English, are formatted with a dollar sign ( \$ ) that precedes the MONEY value, a period ( . ) that separates the integral from the fractional part of the MONEY value, and no *back* symbol. For example, `100.50` is formatted as `$100.50`.

Suppose you want to represent MONEY values as DM (deutsche mark) units, using the currency symbol `DM` and comma ( , ) as the decimal separator. Enter the following command to set the **DBMONEY** environment variable:

```
setenv DBMONEY DM,
```

Here **DM** is the *front* currency symbol that precedes the MONEY value, and a comma separates the integral from the fractional part of the MONEY value. As a result, the value `100.50` is displayed as `DM100,50`.

For more information about how **DBMONEY** formats MONEY values in nondefault locales, see 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 DB-Access 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 143](#).

```
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 DB-Access 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 DB-Access 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 193](#). 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 DB-Access 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.

```
setenvDBPRINTprogram
```

### **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.

```
setenv DBSPACETEMP 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 and HCL Informix® Enterprise Gateway products and by HCL Informix® and by earlier database servers. **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/infmtmp`.

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 166](#) 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.

## DBTIME environment variable

The **DBTIME** environment variable specifies a formatting mask for the display and data-entry format of **DATETIME** values.

The **DBTIME** environment variable is useful in contexts where the **DATETIME** data values to be formatted by **DBTIME** have the same precision as the specified **DBTIME** setting. You might encounter unexpected or invalid display formats for **DATETIME** values that are declared with a different **DATETIME** qualifier.

```
setenvDBTIME' { /literal|%[{ - | o}] [min] [.precision] special} '
```

**literal**

is a literal white space or any printable character.

**min**

is a literal integer, setting the minimum number of characters in the substring for the value that *special* specifies.

**precision**

is the number of digits for the value of any time unit, or the maximum number of characters in the name of a month.

**special**

is one of the placeholder characters that are listed following.

These terms and symbols are described in the pages that follow.

This quoted string can include literal characters and placeholders for the values of individual time units and other elements of a DATETIME value. **DBTIME** takes effect only when you call certain IBM® Informix® ESQ/C DATETIME routines. (For details, see the *HCL® Informix® Enterprise Replication Guide*.) If **DBTIME** is not set, the behavior of these routines is undefined, and "YYYY-MM-DD hh:mm:ss.ffffff" is the default display and input format for DATETIME YEAR TO FRACTION(5) literal values in the default locale.

The percentage (%) symbol gives special significance to the *special* placeholder symbol that follows. Without a preceding % symbol, any character within the formatting mask is interpreted as a literal character, even if it is the same character as one of the placeholder characters in the following list. Note also that the *special* placeholder symbols are case sensitive.

The following characters within a **DBTIME** format string are placeholders for time units (or for other features) within a DATETIME value.

**%b**

is replaced by the abbreviated month name.

**%B**

is replaced by the full month name.

**%d**

is replaced by the day of the month as a decimal number [01,31].

**%Fn**

is replaced by a fraction of a second with a scale that the integer *n* specifies. The default value of *n* is 2; the range of *n* is  $0 \leq n \leq 5$ .

**%H**

is replaced by the hour (24-hour clock).

**%I**

is replaced by the hour (12-hour clock).

**%M**

is replaced by the minute as a decimal number [00,59].

**%m**

is replaced by the month as a decimal number [01,12].

**%p**

is replaced by A.M. or P.M. (or the equivalent in the locale file).

**%S**

is replaced by the second as a decimal number [00,59].

**%y**

is replaced by the year as a four-digit decimal number.

**%Y**

is replaced by the year as a four-digit decimal number. User must enter a four-digit value.

**%%**

is replaced by % (to allow a literal % character in the format string).

For example, consider this display format for DATETIME YEAR TO SECOND:

```
Mar 21, 2013 at 16 h 30 m 28 s
```

If the user enters a two-digit year value, this value is expanded to 4 digits according to the **DBCENTURY** environment variable setting. If **DBCENTURY** is not set, then the string 19 is used by default for the first two digits.

Set **DBTIME** as the following command line (for the C shell) shows:

```
setenv DBTIME '%b %d, %Y at %H h %M m %S s'
```

The default **DBTIME** produces the following ANSI SQL string format:

```
2001-03-21 16:30:28
```

You can set the default **DBTIME** as the following example shows:

```
setenv DBTIME '%Y-%m-%d %H:%M:%S'
```

An optional field width and precision specification (*w.p*) can immediately follow the percent (%) character. It is interpreted as follows:

**w**

Specifies the minimum field width. The value is right-justified with blank spaces on the left.

**-w**

Specifies the minimum field width. The value is left-justified with blank spaces on the right.

**0w**

Specifies the minimum field width. The value is right-justified and padded with zeros on the left.

**p**

Specifies the precision of `d`, `H`, `I`, `m`, `M`, `S`, `y`, and `Y` time unit values, or the maximum number of characters in `b` and `B` month names.

The following limitations apply to field-width and precision specifications:

- If the data value supplies fewer digits than *precision* specifies, the value is padded with leading zeros.
- If a data value supplies more characters than *precision* specifies, excess characters are truncated from the right.
- If no field width or precision is specified for `d`, `H`, `I`, `m`, `M`, `S`, or `y` placeholders, `0.2` is the default, or `0.4` for the `Y` placeholder.
- A *precision* specification is significant only when converting a DATETIME value to an ASCII string, but not vice versa.

The `F` placeholder does not support this field-width and precision syntax.



**Important:** Any separator character between the `%S` and `%F` directives for DATETIME user formats must be explicitly defined. Specifying `%S%F` concatenates the digits that represent the integer and fractional parts of the seconds value.

Like `DBDATE`, `GL_DATE`, or `GL_DATETIME`, or `USE_DTENV`, the `DBTIME` setting controls only the character-string representation of data values. It cannot change the internal storage format of the DATETIME column. (For additional information about formatting DATE values, see the discussion of `DBDATE` in the topic [DBDATE environment variable on page 157](#).)

### DBTIME formats in nondefault locales

If you specify a locale other than U.S. English, the locale defines the culture-specific display formats for DATETIME values. To change the default display format, change the setting of `DBTIME`, or of the `GL_DATETIME` and `USE_DTENV` environment variables.

In East Asian locales that support era-based dates, `DBTIME` can also specify Japanese or Taiwanese eras. See *HCL® Informix® GLS User's Guide* for details of additional placeholder symbols for setting `DBTIME` to display era-based DATETIME values, and for descriptions of the `GL_DATETIME`, `GL_DATE`, and `USE_DTENV` environment variables.

---

#### Related reference

[DATETIME data type on page 93](#)

## 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.

```
setenv DBUPSPACE { 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 **DELIMITED** 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 HCL Informix® 4GL, the DataBlade® API (LIBDMI), and the C++ API, behave as ESQL/C, and use 'n' as the default if no value for DELIMITED is specified on the client system.

- Specifying the DELIMITED environment variable with no value on the client system requires client applications to use the DELIMITED setting that is the default for their application programming interface (API).

```
setenvDELIMITED
```

No value is required; **DELIMITED** takes effect if it exists, and it remains in effect while it is on the list of environment variables. Removing DELIMITED 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 DELIMITED 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.colum3 TO column3;
```

This example is an exception to the general rule that when **DELIMITED** 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 **DELIMITED** to declare storage identifiers that contain characters outside the default SQL character set.

Delimited identifiers are case sensitive. To use delimited identifiers, applications in Informix® ESQL/C must set **DELIMITED** 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 141](#).

**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.

## IFXMONGOAUTH environment variable

Set the **IFXMONGOAUTH** environment variable to enable PAM authentication for MongoDB clients through the wire listener.

You can set the **IFXMONGOAUTH** environment variable to any value or to no value.

```
setenv IFXMONGOAUTH 1
```

Setting the **IFXMONGOAUTH** environment variable is a prerequisite to configuring PAM authentication for MongoDB clients.

You can disable the **IFXMONGOAUTH** environment variable with this command:

```
unsetenv IFXMONGOAUTH
```

### 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:

```
setenv IFX_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.

```
setenv IFX_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 56. Effect of IFX\_EXTDIRECTIVES and EXT\_DIRECTIVES settings on external directives**

	<code>EXT_DIRECTIVES = 0</code>	<code>EXT_DIRECTIVES = 1</code>	<code>EXT_DIRECTIVES = 2</code>
<b>IFX_EXTDIRECTIVES No setting</b>	OFF	OFF	ON
<b>IFX_EXTDIRECTIVES0 = OFF</b>	OFF	OFF	OFF
<b>IFX_EXTDIRECTIVES1 = ON</b>	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  $2.90 \leq \text{version} < 4.0$ , 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 `sysdbclose()`.

## 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_Ln` for levels and `hostname_servernum_Lognnnnnnnnn` 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\_SMX\_TIMEOUT environment variable

Use the IFX\_SMX\_TIMEOUT environment variable to specify the maximum number of seconds for a high-availability replication (HDR), remote stand-alone (RS) or shared disk (SD) secondary server to wait for a message from the primary server in a Server Multiplexer Group (SMX) connection.

```
setenv IFX_SMX_TIMEOUT value
```

### **value**

Any positive numeric value for the number of seconds or `-1` to disable this environment variable. There is no upper limit to the number of seconds that you can specify.

### **default value**

```
10
```

For example, to indicate that the secondary server should wait for no more than 60 seconds, specify:

```
setenv IFX_SMX_TIMEOUT 60
```

If the secondary server does not receive any message after the number of seconds specified in the IFX\_SMX\_TIMEOUT environment variable and after the number of cycles specified in the IFX\_SMX\_TIMEOUT\_RETRY environment variable have completed, the secondary server will print the error message in the **online.log** and close the SMX connection.

If an SMX timeout message is in the **online.log**, you might need to increase the IFX\_SMX\_TIMEOUT value, the IFX\_SMX\_TIMEOUT\_RETRY value, or both of these values and restart secondary node.

This environment variable applies only to secondary servers. If you set this environment variable on the primary server, it will become effective only if the primary server becomes a secondary server after a failure.

---

**Related reference**[IFX\\_SMX\\_TIMEOUT\\_RETRY environment variable on page 184](#)**Related information**[Server Multiplexer Group \(SMX\) connections on page](#)

## IFX\_SMX\_TIMEOUT\_RETRY environment variable

Use the IFX\_SMX\_TIMEOUT\_RETRY environment variable to specify the number of times that the high-availability replication (HDR), remote standalone (RS) or shared disk (SD) secondary server will repeat the wait cycle specified by the IFX\_SMX\_TIMEOUT environment variable if a response from the primary server has not been received.

```
setenv IFX_SMX_TIMEOUT_RETRY value
```

**value**

Any positive numeric value

**default value**

6

For example, to indicate that the amount of time specified in the IFX\_SMX\_TIMEOUT configuration parameter should be repeated up to 20 times if a response from the primary server has not been received, specify:

```
setenv IFX_SMX_TIMEOUT_RETRY 20
```

If the secondary server does not receive any message after the number of seconds specified in the IFX\_SMX\_TIMEOUT environment variable and after the number of cycles specified in the IFX\_SMX\_TIMEOUT\_RETRY environment variable have completed, the secondary server will print the error message in the **online.log** and close the SMX connection. If an SMX timeout message is in the **online.log**, you might need to increase the IFX\_SMX\_TIMEOUT value, the IFX\_SMX\_TIMEOUT\_RETRY value, or both of these values and restart secondary node.

This environment variable applies only to secondary servers. If you set this environment variable on the primary server, it will become effective only if the primary server becomes a secondary server after a failure.

---

**Related reference**[IFX\\_SMX\\_TIMEOUT environment variable on page 183](#)**Related information**[Server Multiplexer Group \(SMX\) connections on page](#)

## IFX\_UNLOAD\_EILSEQ\_MODE environment variable

Use the IFX\_UNLOAD\_EILSEQ\_MODE environment variable to help migrate databases from Informix® Version 10 to Version 11.50 or 11.70, where character data might be encoded with a codeset that is different than the codeset used to create the Version 10 database.

In earlier versions of Informix®, it was possible to load character data into a database that did not match the locale and codeset of the database. For example you could load Chinese data into a database created with the DB\_LOCALE=en\_US.8859-1 codeset. In newer versions of Informix®, to insert Chinese data you would need a database created with the Chinese (DB\_LOCALE=zh\_tw.big5 locale and codeset).

**!** **Important:** For databases created with Version 10 and Client SDK 2.4, when you attempt to unload the invalid character data an error occurs unless you have set this environment variable. The IFX\_UNLOAD\_EILSEQ\_MODE environment variable enables DB-Access, dbexport, and High Performance Loader (HPL) to unload character and bypass the GLS validation that normally occurs when you unload data by using the Version 11.50 and 11.70 tools.

To use this environment variable, set it to any non-empty string.

```
setenv IFX_UNLOAD_EILSEQ_MODE value
```

**value**

Any alpha or numeric value. For example: yes, true, or 1.

This environment variable takes effect when character data is being fetched or retrieved from the database.

```
setenv IFX_UNLOAD_EILSEQ_MODE 1
setenv IFX_UNLOAD_EILSEQ_MODE yes
setenv IFX_UNLOAD_EILSEQ_MODE on
```

This environment variable is similar to setting the EILSEQ\_COMPAT\_MODE configuration parameter in the ONCONFIG file. The configuration parameter affects character data that is inserted into the database, whereas the IFX\_UNLOAD\_EILSEQ\_MODE environment variable affects character data that is unloaded from the database.

## 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.

```
setenv IFX_UPDDESC value
```

**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\_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.

## IMCADMIN environment variable

The **IMCADMIN** environment variable supports the **imcadmin** administrative tool by specifying the name of a database server through which **imcadmin** can connect to MaxConnect. For **imcadmin** to operate correctly, you must set **IMCADMIN** before you use any HCL Informix® products.

```
setenvIMCADMINdbservername
```

**dbservername**

is the name of a database server.

Here *dbservername* must be listed in the **sqlhosts** file on the computer where the MaxConnect runs. MaxConnect uses this setting to obtain the following connectivity information from the **sqlhosts** file:

- Where the administrative listener port must be established
- The network protocol that the specified database server uses
- The host name of the system where the specified database server is located

You cannot use the **imcadmin** tool unless **IMCADMIN** is set to a valid database server name.

For more information about using **IMCADMIN**, see *IBM® Informix® MaxConnect User's Guide*.

## IMCCONFIG environment variable

The **IMCCONFIG** environment variable specifies a nondefault filename, and optionally a pathname, for the MaxConnect configuration file. On UNIX™ systems that support the C shell, this variable can be set by the following command.

```
setenvIMCCONFIGpathname
```

### **pathname**

is a full pathname or a simple filename.

When the setting is a filename that is not qualified by a full pathname, MaxConnect searches for the specified file in the **\$INFORMIXDIR/etc/** directory. Thus, if you set **IMCCONFIG** to **IMCconfig.imc2**, MaxConnect searches for **\$INFORMIXDIR/etc/IMCconfig.imc2** as its configuration file.

If the **IMCCONFIG** environment variable is not set, MaxConnect searches by default for **\$INFORMIXDIR/etc/IMCconfig** as its configuration file.

## IMCSERVER environment variable

The **IMCSERVER** environment variable specifies the name of a database server entry in the **sqlhosts** file that contains information about connectivity.

The database server can be either local or remote. On UNIX™ systems that support the C shell, the **IMCSERVER** environment variable can be set by the command.

```
setenvIMCSERVERdbservername
```

### **dbservername**


is the valid name of a database server.

Here **dbservername** must be the name of a database server in the **sqlhosts** file. For more information about **sqlhosts** settings with MaxConnect, see your *HCL® Informix® Administrator's Guide*. You cannot use MaxConnect unless **IMCSERVER** is set to a valid database server name.

## INFORMIXC environment variable (UNIX™)

The **INFORMIXC** environment variable specifies the filename or pathname of the C compiler to be used to compile files that IBM® Informix® ESQL/C 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 reference**

[INFORMIXCMCONUNITNAME environment variable on page 188](#)

**Related information**

[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 reference

[INFORMIXCMNAME environment variable on page 188](#)

#### Related information

[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.

```
setenv INFORMIXCONCSMCFG 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 `csconfig` 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.

```
setenv INFORMIXCONRETRY 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 190](#)

#### Related reference

[INFORMIXCONRETRY configuration parameter on page](#)

#### Related information

[INFORMIXCONRETRY session environment option 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.

```
setenvINFORMIXCONTIMESeconds
```

**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.

The **INFORMIXCONTIME** and **INFORMIXCONRETRY** environment variables can be modified with the **onutil** SET command, as in the following example:

```
% onutil
1> SET INFORMIXCONTIME 120;
Dynamic Configuration completed successfully
2> SET INFORMIXCONRETRY 10;
Dynamic Configuration completed successfully
```

**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*.

```
setenvINFORMIXDIRpathname
```

#### **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
```

## INFORMIXOPCACHE environment variable

The **INFORMIXOPCACHE** environment variable can specify the size of the memory cache for the staging-area blobspace of the client application.

```
setenvINFORMIXOPCACHEkilobytes
```

**kilobytes**

Specifies the value you set for the optical memory cache.

Set the **INFORMIXOPCACHE** environment variable by specifying the size of the memory cache in KB. The specified size must be equal to or smaller than the size of the system-wide configuration parameter, **OPCACHEMAX**.

If you do not set **INFORMIXOPCACHE**, the default cache size is 128 kilobytes or the size specified in the configuration parameter **OPCACHEMAX**. The default for **OPCACHEMAX** is 0. If you set **INFORMIXOPCACHE** to a value of 0, Optical Subsystem does not use the cache.

## 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 DB-Access 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.

```
setenvINFORMIXSERVERdbservername
```

**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 DB-Access 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`.

```
setenvINFORMIXSHMBASEvalue
```

#### **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.

```
setenvINFORMIXSQLHOSTSpathname
```

#### **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: Windows™:** 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.

```
setenvINFORMIXSTACKSIZEsize
```

### 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 DB-Access 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 DB-Access 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 213](#).

## 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_SEP/n
```

*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`, `dbسادir`, 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).

## ISM\_COMPRESSION environment variable

Use the **ISM\_COMPRESSION** environment variable to specify whether the IBM® Informix® Storage Manager (ISM) should use a data-compression algorithm to store and retrieve data.

```
setenv ISM_COMPRESSION { TRUE | FALSE }
```

If **ISM\_COMPRESSION** is set to `TRUE` in the environment of the ON-Bar process that makes a request, the ISM server uses the data-compression algorithm.

If **ISM\_COMPRESSION** is set to `FALSE` or is not set, the ISM server does not use compression.

## ISM\_DEBUG\_FILE environment variable

Use the **ISM\_DEBUG\_FILE** environment variable in the IBM® Informix® Storage Manager (ISM) server environment to specify where to write XBSA messages.

```
setenv ISM_DEBUG_FILE pathname
```

### *pathname*

Specifies the location of the XBSA message log file.

If you do not set **ISM\_DEBUG\_FILE**, the XBSA message log is located in the `$INFORMIXDIR/ism/appllogs/xbsa.messages` directory on UNIX™, or in the `c:\nsr\appllogs\xbsa.messages` directory on Windows™ systems.

## ISM\_DEBUG\_LEVEL environment variable

Use the **ISM\_DEBUG\_LEVEL** environment variable in the ON-Bar environment to control the level of reporting detail recorded in the XBSA messages log. The XBSA shared library writes to this log.

```
setenv ISM_DEBUG_LEVEL value
```

### *value*

specifies the level of reporting detail, where `1 ≤ value ≤ 9`.

If **ISM\_DEBUG\_LEVEL** is not set, has a null value, or has a value outside this range, the default detail level is `1`. A detail level of `0` suppresses all XBSA debugging records. A detail level of `1` reports only XBSA failures.

## ISM\_ENCRYPTION environment variable

Use the **ISM\_ENCRYPTION** environment variable in the ON-Bar environment to specify whether IBM® Informix® Storage Manager (ISM) uses data encryption.

```
setenv ISM_ENCRYPTION { XOR | NONE | TRUE }
```

Three settings of **ISM\_ENCRYPTION** are supported:

**XOR**

Uses encryption.

**NONE**

Does not use encryption.

**TRUE**

Uses encryption.

If **ISM\_ENCRYPTION** is set to **NONE** or is not set, the ISM server does not use encryption.

If the **ISM\_ENCRYPTION** is set to **TRUE** or **XOR** in the environment of the ON-Bar process that makes a request, the ISM server uses encryption to store or retrieve the data specified in that request.

## ISM\_MAXLOGSIZE environment variable

Use the **ISM\_MAXLOGSIZE** environment variable in the IBM® Informix® Storage Manager (ISM) server environment to specify the size threshold of the ISM activity log.

```
setenv ISM_MAXLOGSIZE size
```

**size**

Specifies the size threshold (in megabytes) of the activity log.

If **ISM\_MAXLOGSIZE** is not set, then the default size limit is 1 megabyte. If **ISM\_MAXLOGSIZE** is set to a null value, then the threshold is 0 bytes.

## ISM\_MAXLOGVERS environment variable

Use the **ISM\_MAXLOGVERS** environment variable in the IBM® Informix® Storage Manager (ISM) server environment to specify the maximum number of activity-log files to be preserved by the ISM server.

```
setenv ISM_MAXLOGVERS value
```

**value**

specifies the number of files to be preserved.

If **ISM\_MAXLOGVERS** is not set, then the default number of files is four. If the setting is a null value, then the ISM server preserves no activity log files.

## 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.

```
setenv JAR_TEMP_PATH pathname
```

***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.

```
setenv JAVA_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.

```
setenv JVM_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.

```
setenv LD_LIBRARY_PATH$PATH: pathname
```

***pathname***

Specifies the search path for the library.

For INTERSOLV DataDirect ODBC Driver on AIX®, set **LIBPATH**. For INTERSOLV DataDirect ODBC Driver 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 INTERSOLV DataDirect ODBC Driver. 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.

```
setenv OPTCOMPIND { 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 OPTCOMPIND 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 IBM® Informix® ESQL/C 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 IBM® Informix® ESQL/C 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 Informix® ESQL/C 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 143](#).

## 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.

## PLCONFIG environment variable

The **PLCONFIG** environment variable specifies the name of the configuration file that the High Performance Loader (HPL) uses. This file must be located in the `$INFORMIXDIR/etc` directory. If the **PLCONFIG** environment variable is not set, then `$INFORMIXDIR/etc/plconfig` is the default configuration file.

```
setenv PLCONFIG filename
```

#### **filename**

Specifies the simple file name of the configuration file that the High-Performance Loader uses.

For example, to specify the `$INFORMIXDIR/etc/custom.cfg` file as the configuration file for the High-Performance Loader, enter the following command:

```
setenv PLCONFIG custom.cfg
```

For more information, see the *IBM® Informix® High-Performance Loader User's Guide*.

## PLOAD\_LO\_PATH environment variable

The **PLOAD\_LO\_PATH** environment variable lets you specify the pathname for smart-large-object handles (which identify the location of smart large objects such as BLOB and CLOB data types).

```
setenvPLOAD_LO_PATHpathname
```

### **pathname**

specifies the directory for the smart-large-object handles.

If **PLOAD\_LO\_PATH** is not set, the default directory is **/tmp**.

For more information, see the *IBM® Informix® High-Performance Loader User's Guide*.

## PLOAD\_SHMBASE environment variable

The **PLOAD\_SHMBASE** environment variable lets you specify the shared-memory address at which the High Performance Loader (HPL) onload processes will attach. If **PLOAD\_SHMBASE** is not set, the HPL determines which shared-memory address to use.

```
setenvPLOAD_SHMBASEvalue
```

### **value**

Used to calculate the shared-memory address.

If the onload utility cannot attach, an error is issued, and you must specify a new value.

The onload utility tries to determine at which address to attach, as follows in the following (descending) order:

1. Attach at the same address (SHMBASE) as the database server.
2. Attach beyond the database server segments.
3. Attach at the address specified in **PLOAD\_SHMBASE**.



**Tip:** It is recommended that you let the HPL decide where to attach and that you set **PLOAD\_SHMBASE** only if necessary to avoid shared-memory collisions between onload and the database server.

For more information, see the *IBM® Informix® High-Performance Loader User's Guide*.

## 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 Informix® Primary Storage Manager 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 Informix® Primary Storage Manager.
- 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.

```
setenv PSM_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.

```
setenv PSM_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.

```
setenv PSORT_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 166](#).

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.

```
setenv PSORT_NPROCthreads
```

***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.

```
setenv RTREE_COST_ADJUST_VALUE value
```

### **value**

is a floating-point number, where  $1 \leq \text{value} \leq 1000$ , 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.

## SHLIB\_PATH environment variable (UNIX™)

The **SHLIB\_PATH** environment variable tells the shell on HP-UX systems which directories to search for dynamic-link libraries. This is used, for example, with the INTERSOLV DataDirect ODBC Driver. You must specify the full pathname for the directory where you installed the product.

```
setenvSHLIB_PATH$PATH: pathname
```

### *pathname*

Specifies the search path for the libraries.

On Solaris systems, set **LD\_LIBRARY\_PATH**. On AIX® systems, set **LIBPATH**.

## 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 DB-Access (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 DB-Access (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 DB-Access 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 DB-Access 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 IBM® Informix® ESQL/C applications. A multithreaded Informix® ESQL/C 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 Informix® ESQL/C script when you compile a multithreaded Informix® ESQL/C application. When you use the `-thread` option while compiling, the Informix® ESQL/C 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 ]
```

### ***tz***

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.

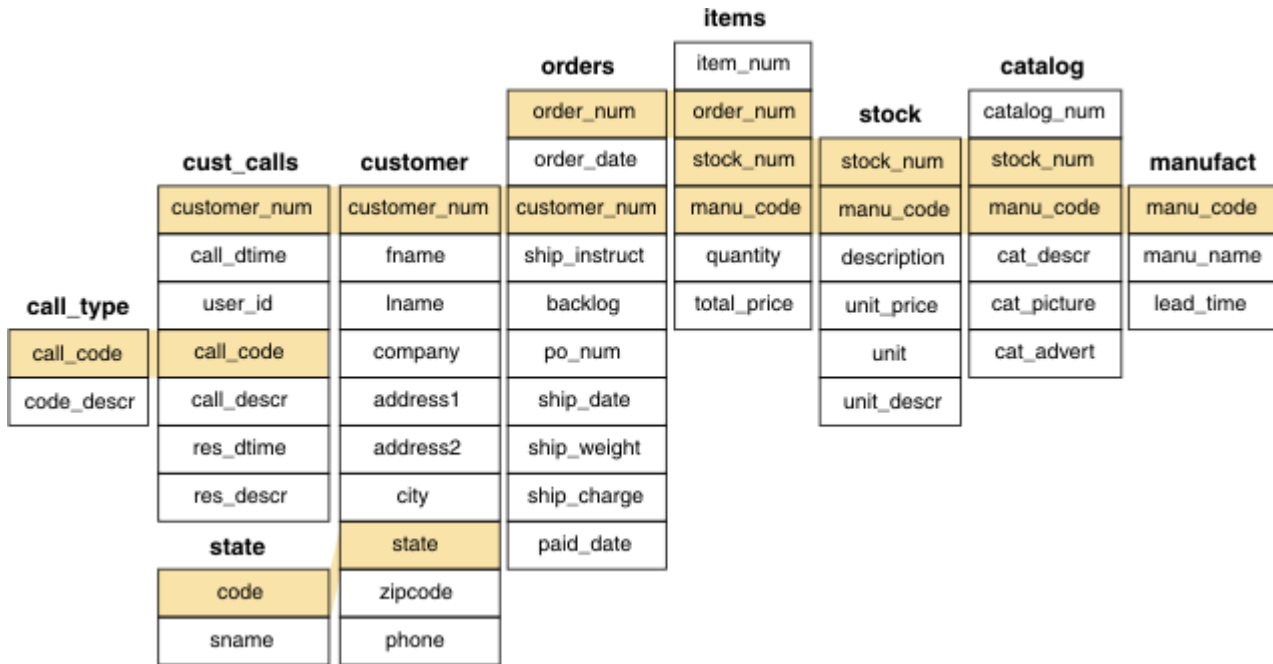
For information about how to create and populate the **stores\_demo** database, see the *HCL® Informix® DB-Access User's Guide*. For information about how to design and implement a relational database, see the *IBM® Informix® Database Design and Implementation Guide*.

### The stores\_demo Database Map

Some of the tables in the **stores\_demo** database have relationships between them.

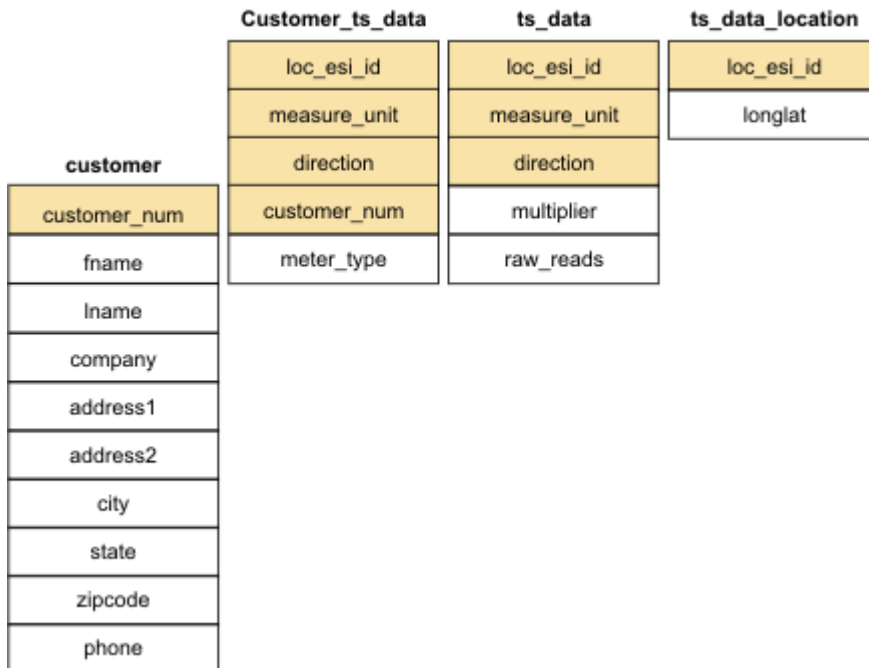
The following illustration displays the joins in the **stores\_demo** database between customers, catalog orders, and customer calls. The shading that connects a column in one table to a column with the same name in another table indicates the relationships, or *joins*, between tables.

Figure 5. Joins between customers and catalog orders



The following illustration displays the joins in the **stores\_demo** database between customers, electricity meter data, and location. The **Customer\_ts\_data**, **ts\_data**, and **ts\_data\_location** tables contain time series data. You can prevent the creation of these time series tables when you create the demonstration database.

Figure 6. Joins between customers, electricity usage data, and location



## 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

<b>location_id</b>	SERIAL
<b>loc_type</b>	CHAR(2)
<b>company</b>	VARCHAR(20)
<b>street_addr</b>	LIST(VARCHAR(25) NOT NULL)
<b>city</b>	VARCHAR(25)
<b>country</b>	VARCHAR(25)

### loc\_us\_t definition

<b>state_code</b>	CHAR(2)
<b>zip</b>	ROW(code INTEGER, suffix SMALLINT)
<b>phone</b>	CHAR(18)

### loc\_non\_us\_t definition

<b>province_code</b>	CHAR(2)
<b>zipcode</b>	CHAR(9)
<b>phone</b>	CHAR(15)

**name\_t definition**

<b>first</b>	VARCHAR(15)
<b>last</b>	VARCHAR(15)

**customer\_t definition**

<b>customer_num</b>	SERIAL
<b>customer_type</b>	CHAR(1)
<b>customer_name</b>	name_t
<b>customer_loc</b>	INTEGER
<b>contact_dates</b>	LIST(DATETIME YEAR TO DAY NOT NULL)
<b>cust_discount</b>	percent
<b>credit_status</b>	CHAR(1)

**retail\_t definition**

<b>credit_num</b>	CHAR(19)
<b>expiration</b>	DATE

**whlsale\_t definition**

<b>resale_license</b>	CHAR(15)
<b>terms_net</b>	SMALLINT

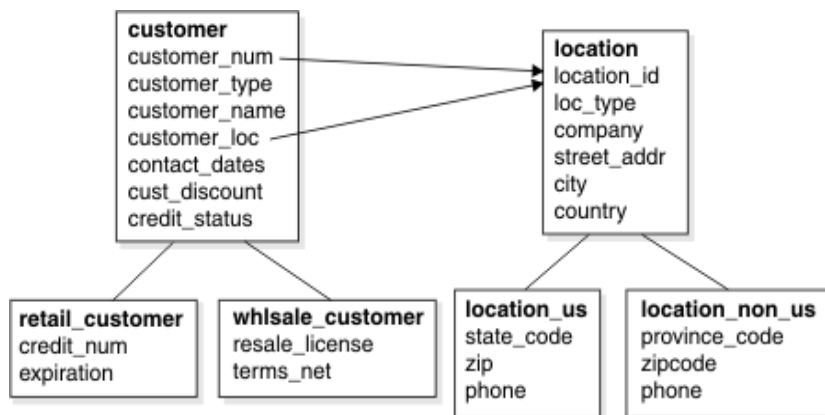
**ship\_t definition**

<b>date</b>	DATE
<b>weight</b>	DECIMAL(8,2)
<b>charge</b>	MONEY(6,2)
<b>instruct</b>	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 7. Hierarchies of superstores\_demo Tables



## Guide to SQL: Tutorial

This publication 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 232](#), describes.

## Illustration of a data model

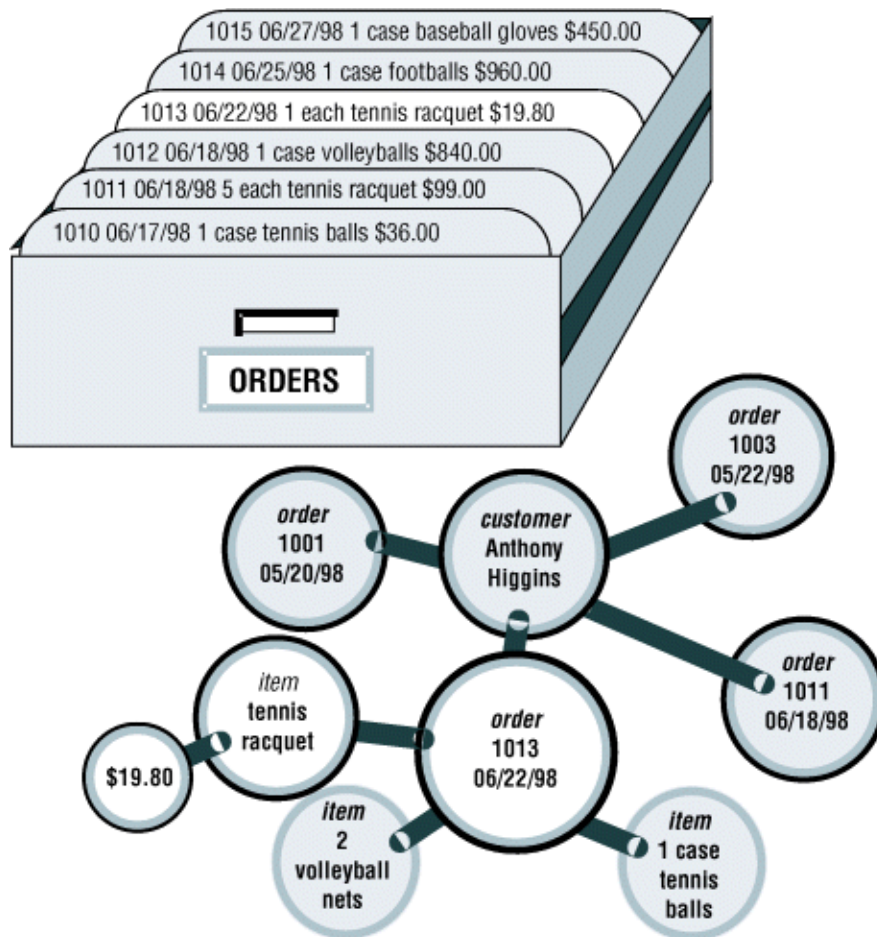
The principal difference between information collected in a database versus information collected in a file is the way the data is organized. A flat file is organized physically; certain items precede or follow other items. But the contents of a database are organized according to a *data model*. A data 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 8. 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.

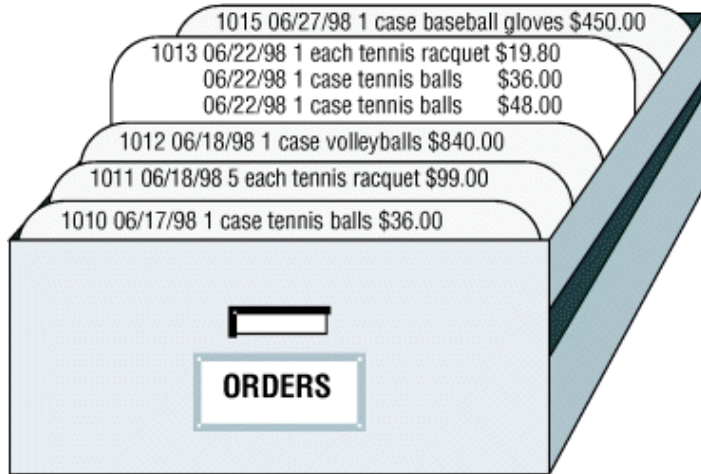
A data model, however, is contained in the database it describes. It travels with the database and is available to any program that uses the database. The model defines not only the names of the data items but also their data types, so a program can adapt itself to the database. For example, a program 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 400](#), and [Modify data through SQL programs on page 423](#).

## Query data

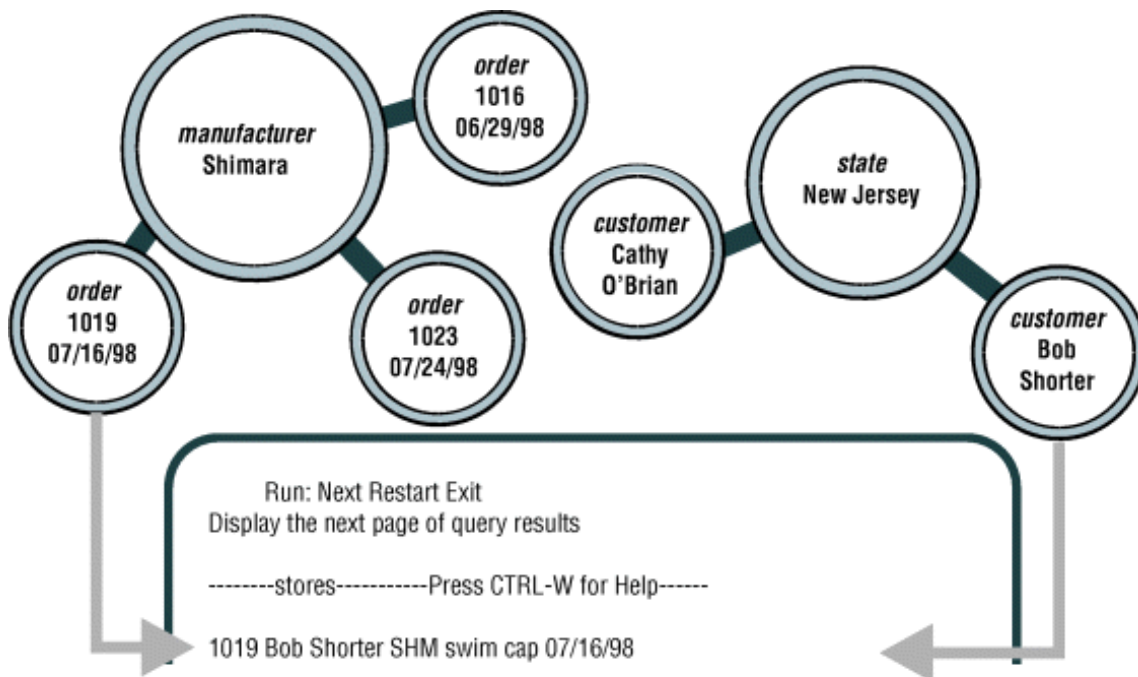
Another difference between a database and a file is the way you can access them. You can search a file sequentially, looking for particular values at particular physical locations in each line or record. That is, you might ask, "What records have the number 1013 in the first field?" The following figure shows this type of search.

Figure 9. 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 10. 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 232](#), and [Compose advanced SELECT statements on page 320](#), 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 358](#).

## 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 HCL Informix® 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.

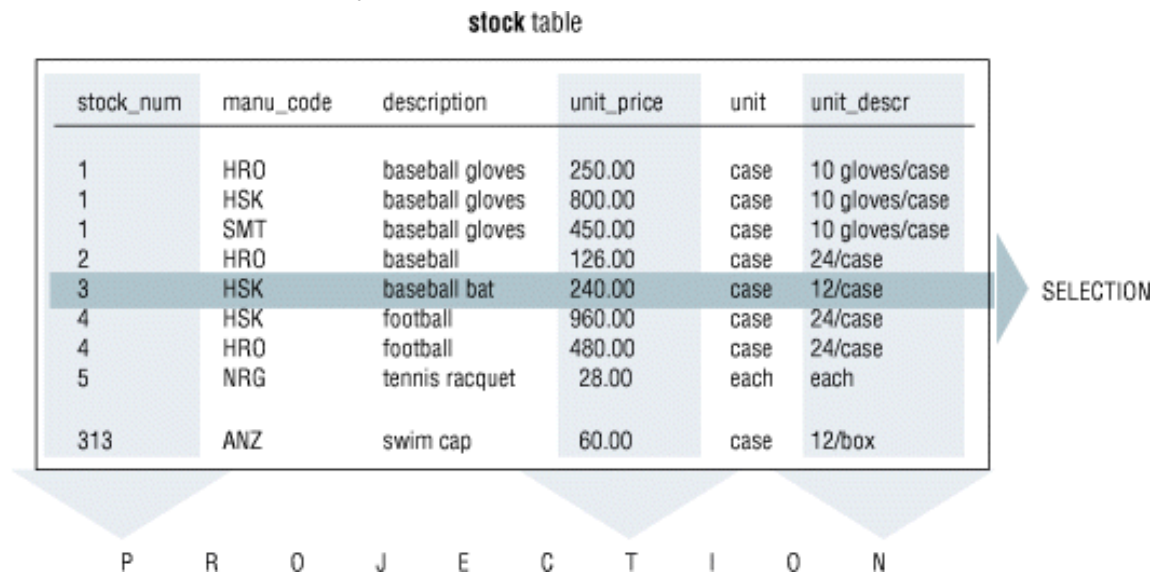
## Sequences

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 11. 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 320](#).

## 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 DataBlade® modules or shared class libraries, that you can add on to the database server, if it suits your needs. A DataBlade® 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 DataBlade® 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 HCL Informix® 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 453](#).

## 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.

## Standard SQL

The relational model and SQL and were invented and developed at IBM® in the early and middle 1970s. Once IBM® proved that it was possible to implement practical relational databases and that SQL was a usable language for manipulating them, other implementations of SQL were developed.

For reasons of performance or competitive advantage, or to take advantage of local hardware or software features, each SQL implementation differed in small ways from the others and from the IBM® version of the language. To ensure that the differences remained small, a standards committee was formed in the early 1980s.

Committee X3H2, sponsored by the American National Standards Institute (ANSI), issued the SQL1 standard in 1986. This standard defines a core set of SQL features and the syntax of statements such as SELECT.

## Informix® SQL and ANSI SQL

The HCL® Informix® implementation of SQL is compatible with standard SQL. Informix® SQL is also compatible with the IBM® version of the language. However, Informix® SQL contains *extensions* to the standard; that is, extra options or features for certain statements, and looser rules for others. Most of the differences occur in the statements that are not in everyday use. For example, few differences occur in the SELECT statement, which accounts for 90 percent of SQL use.

When a difference exists between Informix® SQL and ANSI standard, the *HCL® Informix® Guide to SQL: Syntax* identifies the Informix® syntax as an extension to the ANSI standard for SQL.

## Interactive SQL

To carry out the examples in this book and to experiment with SQL and database design, you need a program that lets you execute SQL statements interactively. DB-Access is such a program. It helps you compose SQL statements and then passes your SQL statements to the database server for execution and displays the results to you.

## 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 DB-Access or an SQL Application Programming Interface (API) such as IBM® Informix® ESQL/C.

[SQL programming on page 400](#), and [Modify data through SQL programs on page 423](#), 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 DB-Access 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 320](#). 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 320](#) display the SELECT statements and their output as they appear when you use the interactive Query-language option in DB-Access.

## Output from large object data types

When you issue a SELECT statement that includes a large object, DB-Access 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

DB-Access 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 358](#). 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 12. 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, DB-Access displays the data from each column on a separate line.

Figure 13. 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 14. 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, DB-Access is able to display all of the data from the row on one line.

Figure 15. 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 16. Query

```
SELECT UNIQUE city, state, zipcode
FROM customer
WHERE state = 'NJ';
```

[Figure 17: Query result on page 236](#) contains a subset of the rows and a subset of the columns in the **customer** table.

Figure 17. Query result

city	state	zipcode
Cherry Hill	NJ	08002
Princeton	NJ	08540

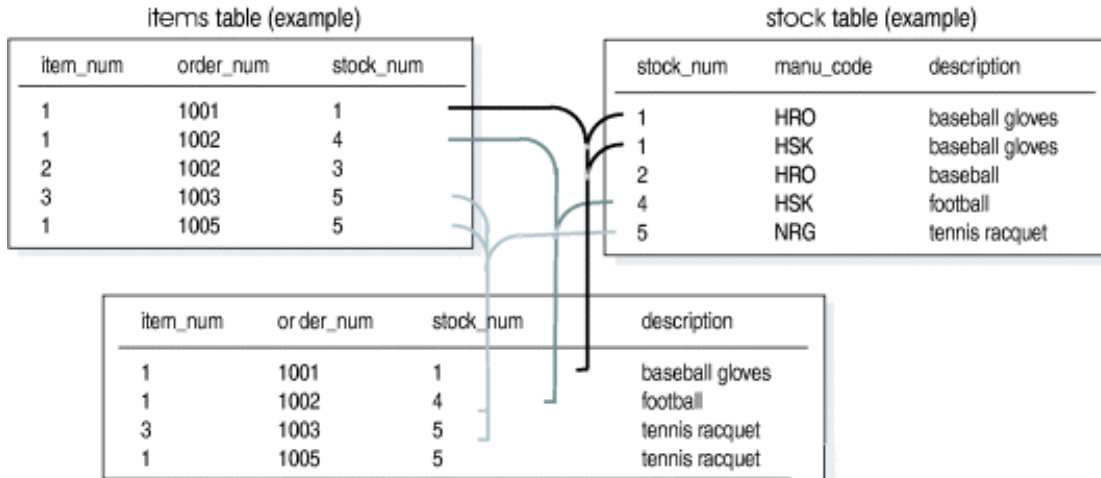
## Join

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 18. A join between two tables

```
SELECT UNIQUE item_num, order_num,
stock.stock_num, description
FROM items, stock
WHERE items.stock_num = stock.stock_num
```



The following query joins the **customer** and **state** tables.

Figure 19. 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 20. 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 21. 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 22. Query

```
SELECT * FROM manufact;
```

Because the **manufact** table has only three columns, [Figure 21: Query on page 238](#) and [Figure 22: Query on page 239](#) 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 23. 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 24. 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 25. 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 15: Query result on page 236](#) and [Figure 25: Query result on page 239](#) 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 26. Query

```
SELECT manu_code, manu_name, lead_time
FROM manufact
ORDER BY lead_time;
```

For HCL Informix®, 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 27. 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 28. 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 29. 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 32: Query on page 241](#) 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 30. 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 31. 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 32. 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 33. 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 34. 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 35. 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 36. 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 37. Query

```
SELECT DISTINCT customer_num FROM orders;

SELECT UNIQUE customer_num FROM orders;
```

To produce a more readable list, [Figure 37: Query on page 243](#) limits the display to show each customer number in the **orders** table only once, as the result shows.

Figure 38. 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 39. Query

```
SELECT po_num FROM orders;
```

The result shows how the statement retrieves data in the **po\_num** column in the **orders** table.

Figure 40. 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 41. Query

```
SELECT po_num FROM orders ORDER BY po_num;
```

Figure 42. 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 43. 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 241](#) 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 44. 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 46: Query result on page 246](#) shows.

Figure 45. 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 46. 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 47. 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 48. 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 49. 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

&lt;

less than

&lt;=

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 HCL Informix®, 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 50. 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 51. 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 52. 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 53. 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	Neele'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 54. 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 55. 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 56. Query

```
SELECT fname, lname, city, state
FROM customer
WHERE zipcode NOT BETWEEN '94000' AND '94999'
ORDER BY state;
```

Figure 57. 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 249](#), 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 58. 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 59. 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 HCL Informix®, 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 58: Query on page 251](#) 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 60. 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 61. 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	Pafo 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 62. Query

```
SELECT order_num, customer_num, po_num, ship_date
FROM orders
WHERE paid_date IS NULL
ORDER BY customer_num;
```



Figure 63. 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 64. 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 65. 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 66. 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 67. 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 396](#) and the *HCL® Informix® Guide to SQL: Syntax*.

Figure 68. 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 69. 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 70. Query

```
SELECT stock_num, manu_code, description, unit_price
FROM stock
WHERE manu_code MATCHES '[A-H]*'
ORDER BY description, manu_code;
```

Figure 71. 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 72. 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 73. 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 74. 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 75. 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 76. 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 77. 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 78. 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 79. 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 80. Query

```
SELECT FIRST 5 * FROM state;
```

Figure 81. 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 82. Query

```
SELECT FIRST 1 * FROM orders;
```

Figure 83. 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 80: Query on page 260](#) except for the ORDER BY clause, returns a different set of rows than [Figure 80: Query on page 260](#).



Figure 84. Query

```
SELECT FIRST 5 * FROM state ORDER BY sname;
```

Figure 85. 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 86. Query

```
SELECT FIRST 10 description, unit_price
FROM stock ORDER BY unit_price DESC;
```

Figure 87. 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 80: Query on page 260](#), except that the SKIP 5 specification instructs the database server to return a different set of rows than [Figure 80: Query on page 260](#).

Figure 88. Query

```
SELECT SKIP 5 FIRST 5 * FROM state ORDER BY sname;
```

Figure 89. 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 HCL Informix®, 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 90. 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 91. 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 92. 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 93. 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 94. Query

```
SELECT customer_num, call_code, call_dtime,
       res_dtime - call_dtime
FROM cust_calls
ORDER BY customer_num;
```

Figure 95. 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 90: Query on page 263](#), [Figure 92: Query on page 263](#), and [Figure 96: Query on page 264](#), 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 96. 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 97. 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 98. 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 99. 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 100. 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 101. 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 HCL Informix®, 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 102. Query

```
SELECT order_num, order_date,
  CASE
    WHEN ship_date IS NULL
    THEN "order not shipped"
  END
FROM orders;
```

Figure 103. 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 377](#).

## 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 104: Query on page 267](#) and [Figure 106: Query on page 267](#) show.

Figure 104. 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 100: Query on page 265](#). 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 105. 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 106. 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 107. Query

```
SELECT rowid, * FROM manufact;
```

Figure 108. 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 109. 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 110. Query result

```

customer_num 101
fname       Ludwig
lname       Pauli
company      All Sports Supplies
address1     213 Erswild 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 Erswild 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 Erswild 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 320](#).

## 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 111. Query

```
SELECT * FROM customer CROSS JOIN state;
```

The results of the query are identical to the results of [Figure 109: Query on page 269](#). 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 269](#). For more information about ANSI syntax, see [ANSI join syntax on page 329](#).

## 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 112. 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 113. 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 114. 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 115. 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 116. 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 117. 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 118. 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 119. 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 118: Query on page 275 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 120. 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 121. 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 120: Query on page 275](#), include specific columns in the projection list to define the SELECT statement more closely, as the following query shows.

Figure 122. 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 123. 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 124. 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 124: Query on page 277](#) with the following query, which does not use aliases.

Figure 125. 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 124: Query on page 277](#) and [Figure 125: Query on page 277](#) are equivalent and retrieve the data that the following query shows.

Figure 126. 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 127. 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 128. 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 396](#) 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 129. 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 130. 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 238](#) 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 268](#) 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 MULTISSET 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 131. 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 132. 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 133. Query

```
SELECT * FROM employee
```

The SELECT statement on the **employee** table returns all rows for all columns.

Figure 134. 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 135. Query

```
SELECT name, address FROM employee
```

Figure 136. 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 137. Query

```
SELECT address FROM employee
```

Figure 138. 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 139. Query

```
SELECT address.city, address.state FROM employee
```

Figure 140. 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 132: SQL syntax that creates the student table. on page 282](#). 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 141. Query

```
SELECT s_address.city, s_address.state FROM student
```

Figure 142. 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 131: SQL syntax that creates the ROW types and table. on page 282.](#)) 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 143. Query

```
SELECT address.zip.z_code FROM employee
```

Figure 144. 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 145. Query

```
SELECT address.* FROM employee;
```

Figure 146. 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 147. 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 358](#).



**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 345](#).

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 148. 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 149. Query

```
SELECT department, direct_reports FROM manager
```

Figure 150. Query result

```
department      marketing
direct_reports  SET {Smith, Waters, Adams, Davis, Kurasawa}

department      engineering
direct_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 148: The manager table on page 286](#)) 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 151. 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 152. 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 153. Query

```
SELECT mgr_name, department
FROM manager
WHERE 'Adams' IN direct_reports
```

Figure 154. 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 155. Query

```
SELECT mgr_name, direct_reports
FROM manager
WHERE 'Adams' IN direct_reports
```

Figure 156. 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 157. Query

```
SELECT mgr_name, department
FROM manager
WHERE 'Adams' NOT IN direct_reports
```

Figure 158. 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 303](#).

## 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 159. 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

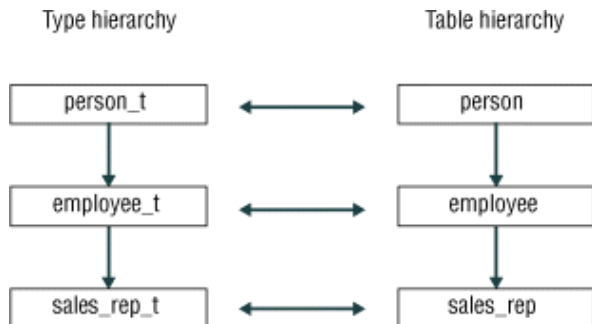
CREATE TABLE person OF TYPE person_t

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 160. 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 161. Query

```
SELECT * FROM person
```

Figure 42: Query result on page 244 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 162. 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 163. Query

```
SELECT * FROM ONLY(person);
```

Figure 164. 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 DB-Access, the following query returns row values for all instances of the **person**, **employee**, and **sales\_rep** tables.

Figure 165. Query

```
SELECT p FROM person p;
```

Informix® ESQL/C does not recognize this construct. In Informix® ESQL/C 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 281](#) 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 286](#) 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 289](#) 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 453](#), 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 320](#). 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 166. Query

```
SELECT AVG (unit_price) FROM stock;
```

Figure 167. 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 168. Query

```
SELECT AVG (unit_price) FROM stock WHERE manu_code = 'SHM';
```

Figure 169. Query result

```
(avg)
$204.93
```

### The COUNT function

The following query counts and displays the total number of rows in the **stock** table.

Figure 170. Query

```
SELECT COUNT(*) FROM stock;
```

Figure 171. 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 172. Query

```
SELECT COUNT (*) FROM stock WHERE manu_code = 'SHM';
```

Figure 173. 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 174. Query

```
SELECT COUNT (DISTINCT manu_code) FROM stock;
```

Figure 175. 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 176. 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 177. 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 178. Query

```
SELECT RANGE(unit_price) FROM stock;
```

Figure 179. 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 180. Query

```
SELECT RANGE(unit_price) FROM stock
GROUP BY manu_code;
```

Figure 181. 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 182. Query

```
SELECT SUM (ship_weight) FROM orders
WHERE ship_date = '07/13/1998';
```

Figure 183. 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:

$$\frac{(\text{SUM}(X_i^{**2}) - (\text{SUM}(X_i)**2)/N)}{(N-1)}$$

In this example,  $X_i$  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 184. 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 185. 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 186. Query

```
SELECT customer_num, DAY (call_dtime), DAY (res_dtime)
FROM cust_calls;
```

Figure 187. 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 188. Query

```
SELECT customer_num, DAY (call_dtime), DAY (res_dtime)
FROM cust_calls
WHERE DAY (call_dtime) < DAY (CURRENT);
```

Figure 189. 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 190. Query

```
SELECT customer_num, call_code, call_descr
FROM cust_calls
WHERE call_dtime < CURRENT YEAR TO DAY;
```

Figure 191. 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 HR0) 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 192. Query

```
SELECT customer_num,
       MONTH (call_dtime) call_month,
       MONTH (res_dtime) res_month
FROM cust_calls;
```

Figure 193. 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 194. Query

```
SELECT customer_num,
       MONTH (call_dtime) called,
       MONTH (res_dtime) resolved
FROM cust_calls
WHERE DAY (res_dtime) < DAY (CURRENT);
```

Figure 195. 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 196. Query

```
SELECT customer_num,
       WEEKDAY (call_dtime) called,
       WEEKDAY (res_dtime) resolved
FROM cust_calls
ORDER BY resolved;
```

Figure 197. 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 198. Query

```
SELECT COUNT(*)
FROM cust_calls
WHERE WEEKDAY (call_dtime) IN (0,6);
```

Figure 199. 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 200. 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 201. 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 202. 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 203. 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 301](#) 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 204. Query

```
SELECT customer_num, call_dtime, res_dtime
FROM cust_calls
WHERE call_dtime > DATE ('12/31/97');
```

Figure 205. 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 206. Query

```
SELECT customer_num,
DATE (call_dtime) called,
DATE (res_dtime) resolved
FROM cust_calls
WHERE call_dtime >= DATE ('1/1/98');
```

Figure 207. 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 208. Query

```
SELECT customer_num,
       TO_CHAR(call_dtime, "%A %B %d %Y") call_date
FROM cust_calls
WHERE call_code = "B";
```

Figure 209. 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 210. Query

```
SELECT order_num,
       TO_CHAR(ship_date,"%A %B %d %Y") date_shipped
FROM orders
WHERE paid_date IS NULL;
```

Figure 211. 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 212. 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 213. 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 214. Query

```
SELECT department, CARDINALITY(direct_reports) FROM manager;
```

Figure 215. 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 216. Query

```
SELECT department, CARDINALITY(direct_reports) FROM manager
WHERE CARDINALITY(direct_reports) < 6
GROUP BY department;
```

Figure 217. 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 57. 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 358](#).

Suppose you create the **inmate** and **fbi\_list** tables, as the following figure shows.

Figure 218. 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 219. Query

```
SELECT manu_code, LOWER(manu_code)
FROM items
WHERE order_num = 1018
```

Figure 220. 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 221. Query

```
SELECT call_code, UPPER(code_descr) FROM call_type
```

Figure 222. 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 223. Query

```
SELECT INITCAP(description) FROM stock
WHERE manu_code = "ANZ";
```

Figure 224. 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 225. Query


```
SELECT stock_num, REPLACE(unit,"each", "item") cost_per, unit_price
FROM stock
WHERE manu_code = "HRO";
```

Figure 226. 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 227. Query

```
SELECT sname, SUBSTRING(sname FROM 1 FOR 4) FROM state
WHERE code = "AZ";
```

Figure 228. 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 229. Query

```
SELECT sname, SUBSTRING(sname FROM 6) FROM state
WHERE code = "WV";
```

Figure 230. 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 231. Query

```
SELECT sname, SUBSTRING(sname FROM -2 FOR 4) FROM state
WHERE code = "AZ";
```



Figure 232. 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 233. Query

```
SELECT sname, SUBSTR(sname, -15, 5) FROM state
WHERE code = "CA";
```

Figure 234. 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 233: Query on page 309](#).

Figure 235. 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 236. Query

```
SELECT sname, LPAD(sname, 21, "-")
FROM state
WHERE code = "CA" OR code = "AZ";
```

Figure 237. 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 238. Query

```
SELECT sname, RPAD(sname, 21, "-")
FROM state
WHERE code = "WV" OR code = "AZ";
```

Figure 239. 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 240. Query

```
SELECT customer_num,
       LENGTH (fname) + LENGTH (lname) namelength
FROM customer
WHERE LENGTH (company) > 15;
```

Figure 241. 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 DB-Access, 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 242. 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 243. 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 244. Query

```
SELECT * FROM orders WHERE order_date = TODAY;
```

Figure 245. 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 246. Query

```
SELECT DBSERVERNAME server, tabid
FROM systables
WHERE tabid <= 4;
```

Figure 247. 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 248. Query

```
SELECT HEX (customer_num) hexnum, HEX (zipcode) hexzip
FROM customer;
```

Figure 249. 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 250. Query

```
SELECT FIRST 1 DBINFO('dbhostname') FROM systables;
```

Figure 251. 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 252. 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 253. Query

```
SELECT emp_id, evaluation FROM employee;
```

Figure 254. 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 255. Query

```
SELECT emp_id, DECODE(evaluation, "poor", 0, "fair", 25, "good",
50, "very good", 75, "great", 100, -1) AS evaluation
FROM employee;
```

Figure 256. 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 257. Query

```
SELECT name, address FROM student;
```

Figure 258. 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 259. Query

```
SELECT name, NVL(address, "address is unknown") AS address
FROM student;
```

Figure 260. 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 453](#).

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 261. Query

```
SELECT customer_num, lname, get_orders(customer_num) n_orders
FROM customer;
```

The result shows the output from this SPL routine.

Figure 262. 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 263. 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 292](#) 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 316](#) shows how to include SPL routines in your SELECT statements.

[Data encryption functions on page 318](#) 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 232](#), 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 278: Query on page 324](#).

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 400](#), 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 232](#) 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 242](#).

Figure 264. Query

```
SELECT DISTINCT customer_num FROM orders;
```

You could also write the statement as the following query shows.

Figure 265. Query

```
SELECT customer_num FROM orders
GROUP BY customer_num;
```

[Figure 264: Query on page 321](#) and [Figure 265: Query on page 321](#) return the following rows.

Figure 266. 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 267. 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 267: Query on page 321](#) 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 268. 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 267: Query on page 321](#) but includes an ORDER BY clause to sort the retrieved rows in ascending order of **price**, as the result shows.

Figure 269. Query

```
SELECT order_num, COUNT(*) number, SUM (total_price) price
FROM items
GROUP BY order_num
ORDER BY price;
```

Figure 270. 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 242](#) 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 269: Query on page 322](#) shows.

Figure 271. 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 272. 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 273. 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 274. Query

```
SELECT order_num, COUNT(*) number, AVG (total_price) average
FROM items
GROUP BY order_num
HAVING COUNT(*) > 2;
```

Figure 275. 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 274: Query on page 323](#), returns just one row, the average of all **total\_price** values in the table, as the result shows.

Figure 276. Query

```
SELECT AVG (total_price) average
FROM items
HAVING count(*) > 2;
```

Figure 277. Query result

average
\$270.97

If [Figure 276: Query on page 324](#), like [Figure 274: Query on page 323](#), 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 278. 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 279. 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 270](#) 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 277](#) 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 280. 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 58. 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 280: Query on page 325](#), labels all columns selected from the **orders** table and puts them in a temporary table called **shipping**.

Figure 281. 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 282. 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 283. 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 284. 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 285. 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 286. 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 335](#).

## Outer joins

This section shows how to create and use outer joins in a SELECT statement. The topic [Create a join on page 270](#) 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 277](#) 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 287. 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 288. 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 is equivalent to the previous ANSI-compliant left outer join:

Figure 289. 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 338](#) 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 290. 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 291. 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 292. 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 293. 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 294. 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 295. 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 296. 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 297. 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 298. 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 299. 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 298: Query on page 334](#) 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 300. 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 301. 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 302. 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 336](#).

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 303. 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 304. 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 305. Query

```
SELECT * FROM (SELECT address.* FROM employee);
```

Figure 306. 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 145: Query on page 286](#) 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 307. 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 308. 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 309. 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 310. 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 311. Query

```
SELECT DISTINCT order_num
  FROM items
 WHERE total_price > ANY
       (SELECT total_price
        FROM items
        WHERE order_num = 1005);
```

Figure 312. 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 313. Query

```
SELECT order_num FROM items
 WHERE stock_num = 9
    AND quantity =
      (SELECT MAX (quantity)
       FROM items
       WHERE stock_num = 9);
```

Figure 314. 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 315. 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 316. 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 317. 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 318. 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 317: Query on page 341](#), 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 319. 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 320. 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 319: Query on page 342 and Figure 320: Query on page 343 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 321. 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 322. 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 323. 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 350](#).

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 324. 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 325. 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 326. 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 327. 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 328. 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 MULTISSET (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 MULTISSET 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 MULTISSET keyword) returns INT values, the collection subquery is of type MULTISSET(INT NOT NULL).

Suppose you create a function `int_func()` that accepts an argument of type MULTISSET(INT NOT NULL). The following query shows a collection subquery that converts rows of INT values to a MULTISSET and uses the collection subquery as an argument in the function `int_func()`.

Figure 329. Query

```
EXECUTE FUNCTION int_func(MULTISSET(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 MULTISSET (INT NOT NULL). Without the ITEM keyword, the collection subquery would return a collection of type MULTISSET (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 346](#) specify collection subqueries by using the TABLE keyword followed (within parentheses) by the MULTISSET 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 MULTISSET 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 330. Query

```
SELECT * FROM TABLE(MULTISSET(SELECT SUM(C1) FROM T1 GROUP BY C1)),
TABLE(MULTISSET(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 331. 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 332. Query

```
SELECT (SELECT c_table.amount FROM TABLE (sales_rep.sales) c_table
        WHERE c_table.month = '98-03')
FROM sales_rep;
```

Figure 333. 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 334. Query

```
SELECT * FROM TABLE((SELECT sales FROM sales_rep
                       WHERE sales_rep.rep_num = 102)) c_table(s_month, s_amount);
```

Figure 335. 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 334: Query on page 348](#) except the derived columns assume the default field names of the **sales** column in the **sales\_rep** table.

Figure 336. Query

```
SELECT * FROM TABLE((SELECT sales FROM sales_rep
WHERE sales_rep.rep_num = 102));
```

Figure 337. 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 348](#) 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 334: Query on page 348](#), and specifies the derived column list **s\_month** and **s\_amount** for the collection-derived table **c\_table**.

Figure 338. Query

```
SELECT * FROM (SELECT sales FROM sales_rep
WHERE sales_rep.rep_num = 102) c_table(s_month, s_amount);
```

Figure 339. 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 336: Query on page 349](#):

Figure 340. Query

```
SELECT * FROM (SELECT sales FROM sales_rep
WHERE sales_rep.rep_num = 102);
```

Figure 341. 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.

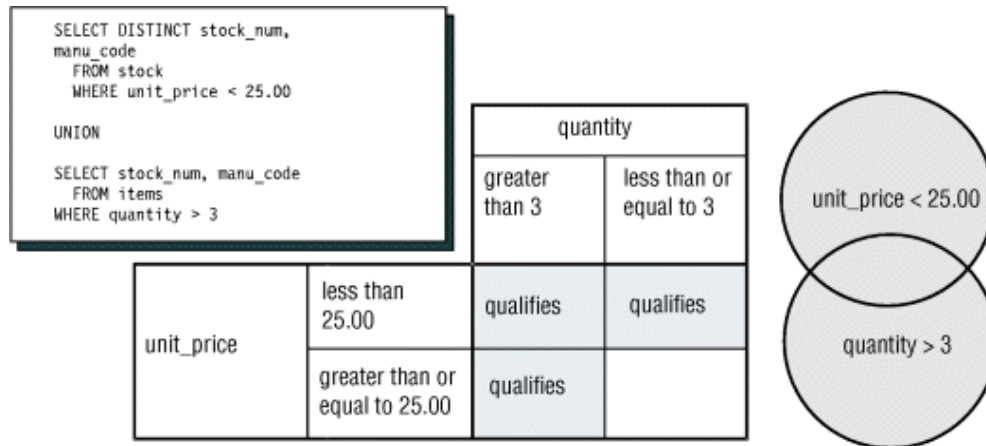
The UNION operator is valid, however, in a subquery in the FROM clause of the SELECT statement, as in the following example:

```
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 342. 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 IBM® Informix® ESQL/C 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 343. 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 344. 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 345. 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 343: Query on page 351](#) but displays them in order of the manufacturer code, as the result shows.

Figure 346. 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 347. 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 345: Query on page 352](#) but also includes duplicate values.

Figure 348. 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 349. 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 350. 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 351. 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 352. 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 353. 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 354. 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 355. 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 356. Query result

```

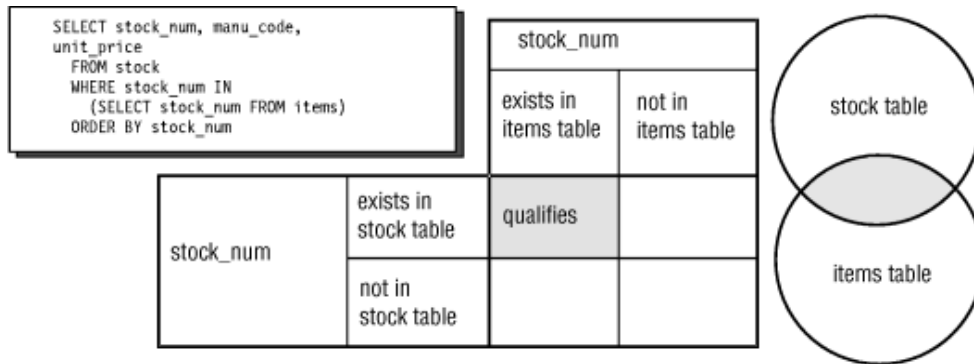
stock_num manu_code
         5 NRG
         5 ANZ
         6 SMT
         6 ANZ
         9 ANZ

```

## 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 357. 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 358. Query

```
SELECT stock_num, manu_code, unit_price FROM stock
WHERE stock_num IN
      (SELECT stock_num FROM items)
ORDER BY stock_num;
```

Figure 359. Query result

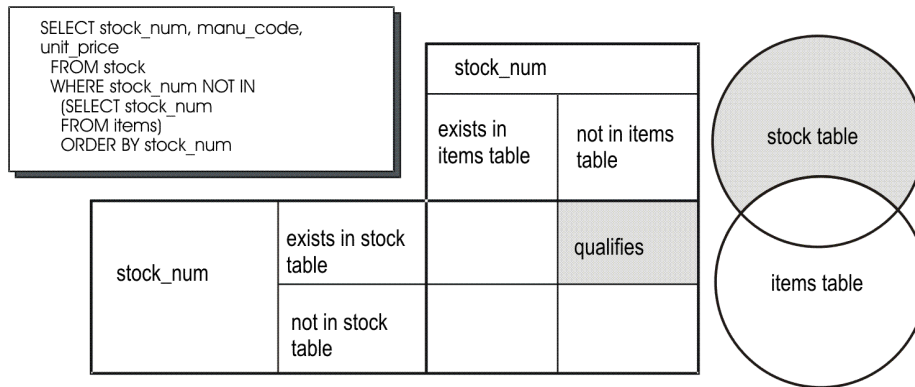
stock_num	manu_code	unit_price
1	HRO	\$250.00
1	HSK	\$800.00
1	SMT	\$450.00
2	HRO	\$126.00
3	HSK	\$240.00
3	SHM	\$280.00
:		
306	SHM	\$190.00
307	PRC	\$250.00
309	HRO	\$40.00
309	SHM	\$40.00

## 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 360. 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 361. 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 362. 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 232](#). 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 MULTISSET 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 391](#). 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 382](#).

## 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 DB-Access 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 392](#).

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 522](#).

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.

HCL 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 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 394](#), and in greater detail in [Programming for a multiuser environment on page 433](#).

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 391](#) 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 380](#).



**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 366](#), 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 363. 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 364: Create the student table. on page 367](#):

```
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 364. 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 Sapphire St.', 'Elmondo', 'CA', '69055'),
  345605900
);
```

## Insert collection values into columns

This section describes how to insert a collection value into a column with DB-Access. 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 Informix® ESQL/C program. For information about how to create Informix® ESQL/C 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 453](#).

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 365. 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 ('sapphire_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 sbpace, 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 304](#).

The following INSERT statement uses the filetoblob() and filetoclob() functions to insert a row of the **inmate** table. ([Figure 218: Create the inmate and fbi\\_list tables. on page 304](#) 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 400](#).) 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 391](#) 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 380](#).



**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 = 'HR0';
```

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 423](#) 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 400](#).

## 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 DB-Access 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 DB-Access. However, you can create Informix® ESQL/C programs and SPL routines to update elements within a collection. For information about how to create Informix® ESQL/C 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 487](#).

## 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 304](#).

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 218: Create the inmate and fbi\_list tables. on page 304 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 362](#)

## 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 **systab** 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:

grantor	grantee	tabid	tabauth
tfecitmutator	101	su-i-x--	
tfecitprocrustes	101	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 223](#) 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 421](#). 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.

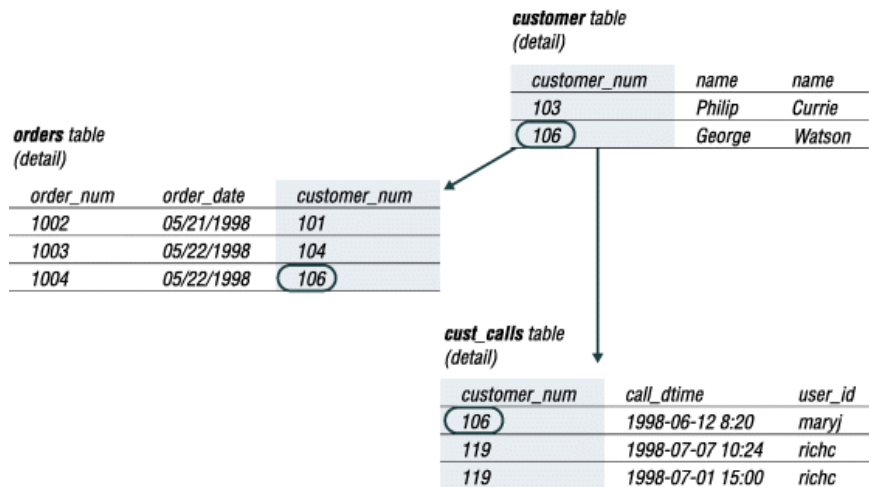
For more information on how to use semantic integrity constraints in database design, see the *IBM® Informix® Database Design and Implementation Guide*.

**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 366. 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 383](#).

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 392](#).

## 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_reowner
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\_reowner** 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.

<b>informix_tupleid</b>	<b>objtype</b>	<b>objowner</b>	<b>objname</b>
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.

<b>informix_tupleid</b>	<b>objtype</b>	<b>objowner</b>	<b>objname</b>
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_reowner
```

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 **Iname** 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\_vio**ls and a diagnostics table named **cust\_subset\_diags** for the **cust\_subset** table, as follows:

```
START VIOLATIONS TABLE FOR cust_subset
USING cust_subset_vio
```

The database server grants the following set of initial privileges on the **cust\_subset\_vio**ls 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 **Iname** column, the **informix\_tupleid** column, the **informix\_optype** column, and the **informix\_reowner** 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\_reowner** 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\_reowner** 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), **Iname** (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 **Iname** 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\_vio**ls and a diagnostics table named **cust\_subset\_diags** for the **cust\_subset** table, as follows:

```
START VIOLATIONS TABLE FOR cust_subset
USING cust_subset_vio
```

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.

HCL Informix® 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 HCL Informix®, 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 *IBM® Informix® Migration Guide*.

If your DBA uses ON-Bar to create backups and back up logical logs, you might also be able to create your own backup copies using ON-Bar. 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 400](#), and [Modify data through SQL programs on page 423](#). For more information about locking and concurrency, see [Programming for a multiuser environment on page 433](#).

## 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® Enterprise Replication 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, MULTISSET, 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, MULTISSET, 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 DB-Access, through application programs written with an SQL API such as Informix® ESQL/C, 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 *ESQL*.

## SQL in SQL APIs

ESQL 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 367. 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 Informix® ESQL/C *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 Informix® ESQL/C application is that each thread can have many active connections to a database server simultaneously. While a nonthreaded Informix® ESQL/C application can establish many connections to one or more databases, it can have only one connection active at a time. A multithreaded Informix® ESQL/C 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 407](#) and [Retrieve multiple rows on page 411](#).

## 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 417](#).

## 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 IBM® Informix® ESQL/C 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 358](#) describes. When the Informix® ESQL/C 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 Informix® ESQL/C, 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 IBM® Informix® ESQL/C, 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 411](#) 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 59: Values of SQLCODE on page 404](#) through [Table 61: Fields of SQLWARN on page 406](#). 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 IBM® Informix® ESQL/C, 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 407](#).

## 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 59. Values of SQLCODE**

Return value	Interpretation
<i>value</i> < 0	Specifies an error code.
<i>value</i> = 0	Indicates success.
0 < <i>value</i> < 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 411](#) 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 integers in the SQLERRD array are set to different values following different statements. The first and fourth elements of the array are used only in IBM® Informix® ESQL/C. The following table shows how the fields are used.



**Table 60. 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 ISAM 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. (DB-Access 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 61. Fields of SQLWARN**

Field	When opening or connecting to a database	All other SQL operations
First	Set to <code>w</code> when any other warning field is set to <code>w</code> . If blank, others need not be checked.	Set to <code>w</code> when any other warning field is set to <code>w</code> .
Second	Set to <code>w</code> when the database now open uses a transaction log.	Set to <code>w</code> 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 <code>w</code> when not all seven table-level privileges are revoked.
Third	Set to <code>w</code> when the database now open is ANSI compliant.	Set to <code>w</code> when a FETCH or SELECT statement returns an aggregate function (SUM, AVG, MIN, MAX) value that is NULL.
Fourth	Set to <code>w</code> when the database server is HCL Informix®.	On a SELECT ... INTO, FETCH ... INTO, or EXECUTE ... INTO statement, set to <code>w</code> 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 <code>w</code> when not all seven table-level access privileges are granted.
Fifth	Set to <code>w</code> when the database server stores the FLOAT data type in DECIMAL form (done when the host system lacks support for FLOAT types).	Set to <code>w</code> after a DESCRIBE statement if the prepared object contains a DELETE statement or an UPDATE statement without a WHERE clause.
Sixth	Reserved.	Set to <code>w</code> following execution of a statement that does not use ANSI-standard SQL syntax (provided the <b>DBANSIWARN</b> environment variable is set).
Seventh	Set to <code>w</code> 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 <code>w</code> when a data fragment (a dbspace) has been skipped during query processing (when the DATASKIP feature is on).
Eighth	Set to <code>w</code> when client DB_LOCALE does not match the database locale. For more information, see the <i>HCL® Informix® GLS User's Guide</i> .	Set to <code>w</code> 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 IBM® Informix® ESQL/C, 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 411](#).

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 IBM® Informix® ESQL/C:

```
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 232](#) or [Compose advanced SELECT statements on page 320](#). 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 Informix® ESQL/C 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 Informix® ESQL/C 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 IBM® Informix® ESQL/C 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 Informix® ESQL/C library function to assign a default value to the host variable. (The function `rstrdate` is part of the IBM® Informix® ESQL/C 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 423](#) 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 *freed* 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 IBM® Informix® ESQL/C:

```
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 423](#) 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 413](#) 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 413](#)). 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 423](#) 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 Informix® ESQL/C:

```
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 IBM® Informix® ESQL/C 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 IBM® Informix® ESQL/C.

```

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 IBM® Informix® ESQL/C 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 423](#) 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 368. 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 IBM® Informix® ESQL/C:

```
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.

DB-Access, a utility that you can use to explore SQL interactively, is the IBM® Informix® ESQL/C program that constructs, prepares, and executes SQL statements dynamically. For example, DB-Access lets you use simple, interactive menus to specify the columns of a table. When you are finished, DB-Access 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 IBM® Informix® ESQL/C:

```
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 405](#)).

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 IBM® Informix® ESQL/C:

```
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 IBM® Informix® ESQL/C 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 IBM® Informix® ESQL/C, 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 DB-Access. 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 IBM® Informix® ESQL/C 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 223](#) and [Privileges on a database and on its objects on page 378](#). 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 400](#), 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 358](#). 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 358](#). 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 IBM® Informix® ESQL/C:

```
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 433](#), 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 423](#), 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 IBM® Informix® ESQL/C 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 Informix® ESQL/C 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 433](#). 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 433](#) 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 429](#) 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 358](#) 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.

The following 4GL code fragment shows the declaration of an insert cursor:

```
DEFINE the_company LIKE customer.company,
       the_fname LIKE customer.fname,
       the_lname LIKE customer.lname
DECLARE new_custs CURSOR FOR
  INSERT INTO customer (company, fname, lname)
    VALUES (the_company, the_fname, the_lname)
```

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 427](#)) 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 427](#). 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 425](#) 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 IBM® Informix® ESQL/C 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_paidddate;
    } 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 425](#). 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 358](#) 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 IBM® Informix® ESQL/C 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 433](#) 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 IBM® Informix® ESQL/C 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 358](#) 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_cursor 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.

---

**L  
ock  
t  
ype**
**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.

**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.

**Exclusive** 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. (or 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.

**Update**  
(or  
update)

---

## 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 450](#).

## 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 449](#).

## 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 435 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 400](#) 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

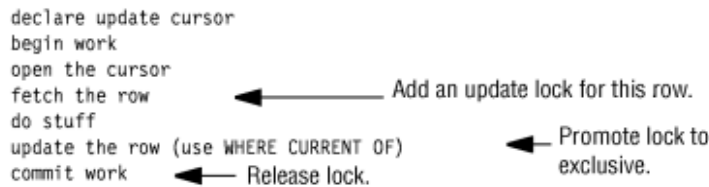
An update cursor is a special kind of cursor that applications can use when the row might potentially be updated. To use an update cursor, execute `SELECT FOR UPDATE` in your application. Update cursors use *promotable locks*; that is, the database server places an update lock (meaning other users can 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.

**i** **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 369. 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

	Hold X lock	Hold U lock	Hold S lock	Hold IS lock	Hold SIX lock	Hold IX lock
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 IBM® Informix® ESQL/C. The CREATE PROCEDURE and CREATE FUNCTION statements are available with DB-Access.

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 370. 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 371. 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 372. 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 373. 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 374. 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 375. 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 376. 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, if 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 377. 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 483](#).

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 378. 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 379. 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 380. 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 232](#).


## 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 381. 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.

The following figure shows how to use the WITH clause of a CREATE FUNCTION statement to specify a negator function.

Figure 382. 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 383. 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 384. 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 385. 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 386. 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 Informix® ESQL/C program:

```
/* This program creates whatever routine is in *
 * the file 'read_add_source'.
 */
#include <stdio.h>
EXEC SQL include sqlca;
EXEC SQL include sqllda;
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 387. 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 388. 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 389. 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 VARCHAR. 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 390. 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 472](#).

## 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 391. 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 392. 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 393. 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 394. 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, MULTISSET, 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, MULTISSET, 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 395. 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 396. 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 397. 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 398. 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 399. 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 400. 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 401. 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 402. 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 403. 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 404. 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 405. 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 406. 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 407. 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 408. Global variable default values.

```
EXECUTE FUNCTION func2();
EXECUTE FUNCTION func1();
```

For more information, see [Executing routines on page 505](#).

## 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 409. 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 396: Named and unnamed row variables. on page 468](#) 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 410. 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 411. 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 486](#) 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 412. 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 413. 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 414. 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 415. 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 416. 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 417. 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 418. 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 419. 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 420. 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 l=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 421. 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 422. 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 423. 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 303](#).

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 424. 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 425. 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 426. 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 427. 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 428. 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 429. 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 430. 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 468](#).

The following figure defines a row type **salary\_t** and an **emp\_info** table, which are the examples that this section uses.

Figure 431. 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 432. 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 433. 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 434. 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 435. 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 436. 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 437. 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 438. 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 435: Declare a collection variable. on page 489](#) 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 MULTISSET

To insert into a `SET` or `MULTISSET` stored in a collection variable, use an `INSERT` statement and follow the `TABLE` keyword with the collection variable, as the following figure shows.

Figure 439. Insert into a `SET` or `MULTISSET` 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 345](#). 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 440. 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 441. 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 `MULTISSET`.



**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 `MULTISSET`, 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 442. 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 443. 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 444. 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 445. FOREACH and END FOREACH statements.

```

FOREACH cursor1 FOR
. . .
END FOREACH

```

The FOREACH statement is described in [The FOREACH loop on page 476](#) and the *HCL® Informix® Guide to SQL: Syntax*.



The next topic, [The collection query on page 493](#), 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 434: Manipulate collections. on page 488](#).

## 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 446. Collection-derived table.

```
FOREACH cursor1 FOR
    SELECT * INTO pnt FROM TABLE(vertices)
    . . .
END FOREACH
```

The SELECT statement uses the collection variable **vertices** 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 **vertices** as a table with four rows, such as the one that the following figure shows.

Figure 447. 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 **vertices** and stores it in **pnt**, so that **pnt** contains the value `'(3.0,1.0)'`.



**Tip:** Because the collection variable **vertices** contains a SET, not a LIST, the elements in **vertices** 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 448. 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 497](#) and [Delete a collection element on page 494](#) 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 449. 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 496](#).

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 450. 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 451. 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 452. 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 499](#) 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 452: Update the collection element. on page 497](#) 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 422: Define the manager table. on page 479](#) defines the **manager** table.

Figure 453. 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 454. 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 422: Define the manager table. on page 479](#) has a column named **projects** that contains a LIST of ROW types with the definition that the following figure shows.

Figure 455. 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 456. 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 497](#) 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 457. 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 458. 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 459. 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, MULTISSET, 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 358](#).

## 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 433: Handle collections in SPL programs. on page 488](#) 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 460. 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 461. 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 462. 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 463. 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 464. 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 465. 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 466. 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 504](#), [2 on page 504](#), and [3 on page 504](#). 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 467. 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 DB-Access
- 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.

**i 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 DB-Access 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 468. 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 469. 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 470. 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 471. 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 472. 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 473. 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 474. 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 475. 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 476. 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 477. 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 514](#).

- 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 DataBlade® 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 DataBlade® 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 DB-Access, 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 DB-Access, 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 461](#).

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 478. 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
- ISAM 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 479. 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 480. 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 481. 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 482. 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 483. 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 484. 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 485. 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:

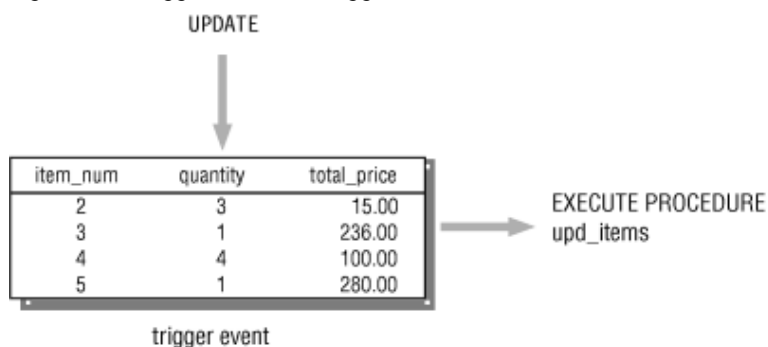
- Create an audit trail of activity in the database. For example, you can track updates to the orders table by updating corroborating information to an audit table.
- Implement a business rule. For example, you can determine when an order exceeds a customer's credit limit and display a message to that 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 486. 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 526](#).

To create a trigger, use DB-Access or one of the SQL APIs. This section describes the CREATE TRIGGER statement as you enter it with the interactive Query-language option in DB-Access. 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 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 529](#).

## 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 453](#).

## 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.

## Apply a fixed error message

You can apply error number -745 to any trigger failure that is not an SQL error. The following fixed message is for this error:

```
-745 Trigger execution has failed.
```

You can apply this message with the RAISE EXCEPTION statement in SPL. The following example generates error -745 if **new\_qty** is greater than **old\_qty** multiplied by 1.50:

```
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 -745;
  END IF
END PROCEDURE
```

If you are using DB-Access, the text of the message for error -745 displays on the bottom of the screen, as the following figure shows.

Figure 487. Error message -745 with fixed message

```
Press CTRL-W for Help
SQL: New Run Modify Use-editor Output Choose Save Info Drop Exit
Modify the current SQL statements using the SQL editor.

----- stores8@myserver ----- Press CTRL-W for Help -----

INSERT INTO items VALUES( 2, 1001, 2, 'HRO', 1, 126.00);

745: Trigger execution has failed.
```

If your trigger calls a procedure that contains an error through an SQL statement in your SQL API, the database server sets the SQL error status variable to -745 and returns it to your program. To display the text of the message, follow the procedure that your HCL® Informix® application development tool provides for retrieving the text of an SQL error message.

## Generate a variable error message

Error number -746 allows you to provide the text of the error message. Like the preceding example, the following one also generates an error if **new\_qty** is greater than **old\_qty** multiplied by 1.50. However, in this case the error number is -746, and the message text `Too many items for Mfr.` is supplied as the third argument in the `RAISE EXCEPTION` statement. For more information on the syntax and use of this statement, see the `RAISE EXCEPTION` statement in [Create and use SPL routines on page 453](#).

```
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, 'Too many items for Mfr.';
  END IF
END PROCEDURE;
```

If you use DB-Access to submit the triggering statement, and if **new\_qty** is greater than **old\_qty**, you will get the result that the following figure shows.

Figure 488. Error Number -746 with User-Specified message Text

```
Press CTRL-W for Help
SQL:  New Run Modify Use-editor Output Choose Save Info Drop Exit
Modify the current SQL statements using the SQL editor.

----- store7@myserver ----- Press CTRL-W for Help -----

INSERT INTO items VALUES( 2, 1001, 2, 'HR0', 1, 126.00);

746: Too many items for Mfr.
```

If you invoke the trigger through an SQL statement in an SQL API, the database server sets **sqlcode** to `-746` and returns the message text in the **sqlerrm** field of the SQL communications area (SQL;CA). For more information about how to use the SQL;CA, see your SQL API publication.

## 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
  - substring operator 89, 139
- !=, not equal, relational operator 249
- ?, question mark
  - as placeholder in PREPARE 418
- 'VERSION' table 65
- (\_), underscore
  - in SQL identifiers 172
- (;), semicolon
  - list separator 192, 204
- (:), colon
  - cast (::) operator 137, 139
  - DATETIME delimiter 93
  - INTERVAL delimiter 101
  - list separator 166, 174, 192, 199, 204
- (=), not equal to
  - relational operator 139
- ('), single quotation
  - string delimiter 162
- ('), single quotation symbols
  - string delimiter 172
- ("), double quotation marks
  - string delimiter 103
- ("), double quotation symbols
  - delimited SQL identifiers 172
  - string delimiter 81, 106, 113
- ( ), parentheses
  - delimiters in expressions 127
- { }, braces
  - collection delimiters 103, 106
  - pathname delimiters 143
- (/), slash
  - DATE separator 92, 127, 157
  - division operator 123, 139
  - pathname delimiter 145, 163, 199
- (\), backslash
  - invalid as delimiter 159
  - pathname delimiter 146, 194
- (#), sharp
  - comment indicator 141
- (%), percentage
  - DBTIME escape symbol 167
  - pathname indicator 166
- (<), less than
  - angle (<>) brackets 89
  - relational operator 139, 159
- (>), greater than
  - angle (<>) brackets 89
  - relational operator 9, 139
- (|), vertical bar
  - absolute value delimiter 100
  - concatenation (||) operator 139
  - field delimiter 159
- (\$), dollar sign
  - currency symbol 105, 162
  - pathname indicator 204
- (-), hyphen
  - DATE separator 157
  - DATETIME delimiter 93
  - INTERVAL delimiter 101
  - subtraction operator 123, 139
  - symbol in syscolauth 4, 22
  - symbol in sysfragauth 36
  - symbol in systabauth 64
  - unary operator 124, 139
- (,), comma

- decimal point 162
- list separator 106, 110, 166
- thousands separator 105
- (.), period
  - DATE separator 157, 157
  - DATETIME delimiter 93
  - decimal point 97, 105, 162
  - execution symbol 141
  - INTERVAL delimiter 101
  - membership operator 139
  - nested dot notation 131
- () , blank space
  - DATETIME delimiter 93
  - INTERVAL delimiter 101
  - padding CHAR values 91
  - padding VARCHAR values 117
- (\*), asterisk
  - multiplication operator 87, 123, 128, 139
  - sysabauth value 4, 64
  - wildcard symbol 20, 77
- (+), plus sign
  - addition operator 123, 139
  - truncation indicator 179
  - unary operator 139
- (=), equality
  - assignment operator 146
  - relational operator 20, 87, 91, 139
- (~), tilde
  - pathname indicator 145
- =, equals, relational operator 248, 271
- >=, greater than or equal to, relational operator 249

## A

- Abbreviated year values 93, 154, 156, 157, 167
  - ACCESS keyword 122
  - Access method
    - B-tree 14, 14, 41, 172
    - built-in 14, 14
    - primary 14, 63, 63
    - R-Tree 172
    - secondary 14, 28, 43, 108
    - sysams data 14
    - sysindices data 43
    - sysopclasses data 48
    - systabamdata data 63
  - Access modes, description of 449
  - Access privilege on UDRs 511
  - ACCESS\_METHOD keyword 14
  - Active set
    - definition of 247, 407
    - of a cursor 414
  - Activity-log files 198
  - Addition (+) operator 123, 139
  - Administrative listener port 186
  - Aggregate functions 115
    - and GROUP BY clause 321
    - AVG 293
    - built-in 103, 106, 113
    - COUNT 293
    - description of 293, 301
    - finding NULL values 410
    - in ESQ 407
    - in expressions 292
    - in SPL routine 475
    - in subquery 340
    - MAX 294
    - MIN 294
  - no BYTE argument 89
  - no collection arguments 103, 106, 113
  - null value signalled 405
  - RANGE 294
  - standard deviation 295
  - STDEV 295
  - SUM 296
  - sysaggregates data 13
  - user-defined 13
  - VARIANCE 296
- AIX operating system 178, 200
- Alias
  - for table name 277
  - to assign column names in temporary table 325
  - using
    - as a query shortcut 277
    - with a supertable 291
    - with self-join 325
- Alias of a table 4
- Alignment of data type 75
- Alignment of data types 20
- ALL keyword
  - beginning a subquery 339
  - in subquery 339
- ALL operator 139
- ALTER INDEX statement, locking table 437
- ALTER OPTICAL CLUSTER statement 48
- Alter privilege 4, 64, 78
- ALTER SEQUENCE statement 215
- ALTER TABLE statement
  - casting effects 134
  - changing data types 81
  - lock mode 175
  - next extent size 9
  - SERIAL columns 111
  - SERIAL8 columns 113
  - synonyms 215
- am\_beginscan() function 14
- am\_close() function 14
- am\_getnext() function 14
- am\_insert() function 14
- am\_open() function 14
- AND logical operator 253
- AND operator 20, 139
- ANSI
  - isolation levels 446
  - SQL version 230
- ANSI compliance
  - ansi flag 153
  - DATETIME literals 167
  - DBANSIWARN environment variable 153
  - DECIMAL range 97
  - DECIMAL(p) data type 96
  - Information Schema views 76
  - isolation level 80
  - public synonyms 63, 65
- ANSI standard
  - as extension to Informix syntax 230
- ANSI-compliant database
  - FOR UPDATE not required in 432
  - signalled in SQLWARN 405
- ANSIOWNER environment variable 148
- ANY keyword, in SELECT statement 339
- ANY operator 139

- Application
  - handling errors 409
  - isolation level 442
  - update cursor 446
- Arabic locales 90
- Archiving
  - database server methods 393
  - description of 393
  - setting DBREMOTECMD 165
  - transaction log 393
- Arithmetic
  - DATE operands 92, 126
  - DATETIME operands 124
  - integer operands 87, 100, 100, 115
  - INTERVAL operands 101, 125
  - operators 139
  - string operands 90
  - time operands 123
- Arithmetic expressions 262
- Arithmetic operators, in expression 262
- AS keyword 137, 137
- Ascending order in SELECT 240
- assign() support function 129
- Asterisk notation, in a SELECT statement 285
- Asterisk, wildcard character in SELECT 238
- AT keyword 103
- Attached index 172
- Attached indexes 39, 156, 210
- Audit Analysis officer 195, 195
- Authorization identifier 71, 80, 364
- AUTO\_STAT\_MODE configuration parameter 31, 37
- AUTO\_STAT\_MODE session environment setting 31, 37
- AVG function, as aggregate function 293

## B

- B-tree access method 14, 41, 172
- B-tree index 39
- Backslash ( \ ) symbol 159
- Backup
  - file prefix 182
- Bandwidth 182
- BEGIN WORK statement 393
- BETWEEN keyword
  - using in WHERE clause 247
  - using to specify a range of rows 249
- BETWEEN operator 139
- BIGINT data type 86
  - coltype code 23
  - length (syscolumns) 27
- BIGSERIAL data type 87
  - coltype code 23
  - last BIGSERIAL value inserted 313
  - length (syscolumns) 27
- bin subdirectory 143
- Binding style 79
- BLOB data type
  - casting unavailable 87
  - defined 87
  - inserting data 87
  - syscolattrs data 21
- Blobspaces
  - defined 122
  - memory cache for staging 192
  - names 172
  - sysblobs data 19
- BOOLEAN data type
  - defined 88
- Boolean expression 253
  - with BOOLEAN data type 88

- with BYTE data type 89
- Boolean expression with TEXT data type 115
- Borland C compiler 187
- Bourne shell 141, 142
- Braces ( { } ) comment delimiters 462
- Bracket ( [ ] ) symbols 115
- brackets substring 115
- Buffers
  - BYTE or TEXT storage (DBBLOBBUF) 154
  - fetch buffer (FET\_BUFFER\_SIZE) 174
  - fetch buffer (SRV\_FET\_BUFFER\_SIZE) 211
  - floating-point display (DBFLTMASK) 160
  - network buffer (IFX\_NETBUF\_SIZE) 180
  - private network buffer pool 180
- Built-in access method 14, 14
- Built-in aggregates 13, 103, 106, 113
- Built-in casts 20, 134
- Built-in data type, declaring variables 467
- Built-in data types
  - casts 134, 138
  - listed 118
  - syscolumns.coltype code 23
  - sysdistrib.type code 31
  - sysxdtypes data 75
- Built-in opaque data types 137
- BY clause 115
- BY keyword 89, 115
- BY ORDER 115
- BYTE data type
  - casting to BLOB 89
  - coltype code 23
  - defined 89
  - increasing buffer size 154
  - inserting values 89
  - restrictions
    - in Boolean expression 89
    - sysables.npused 65
    - with GROUP BY 89
    - with LIKE or MATCHES 89
    - with ORDER BY 89
  - restrictions with GROUP BY 321
  - selecting from BYTE columns 89
  - setting buffer size 154
  - sysblobs data 19, 19
  - syscolumns data 27
  - sysfragments data 39
  - sysopclstr data 48, 48
  - using LENGTH function on 311
  - with relational expression 247

## C

- C compiler
  - default name 187
  - INFORMIXC setting 187
  - thread package 214
- C shell 141
  - .cshrc file 142
  - .login file 142
- C++ map file 192
- CALL statement, in SPL function 507
- Cardinality function
  - description of 303
- CARDINALITY function 303
- CARDINALITY() function 103, 106, 113
- Cartesian product
  - basis of joins 270
  - description of 269
- Cascading deletes 56
  - child tables 383
  - definition of 383
  - locking associated with 383

- logging 383, 392
- referential integrity 383
- restriction 384
- Cascading Select trigger 531
- Case conversion
  - with INITCAP function 306
  - with LOWER function 305
  - with UPPER function 306
- CASE expression
  - description of 265
  - in UPDATE statement 377
  - using 265
- Case-insensitive databases 10, 107, 107
- Cast ( :: ) operator 137, 139
- CAST AS keywords 137
- casting to CLOB 115
- Casts 133, 138
  - built-in 20, 134, 137
  - distinct data type 138
  - explicit 20, 137, 137
  - from BYTE to BLOB 89
  - implicit 20, 137, 137
  - rules of precedence 137
  - syscasts data 20
  - user-defined (UDCs) 20
- Casts from TEXT 115
- CHAR data type
  - built-in casts 136
  - collation 90, 90
  - converting to a DATE value 301
  - converting to a DATETIME value 302
  - defined 90
  - in relational expressions 247
  - nonprintable characters 91, 91
  - storing numeric values 90
  - substrings of 246
  - truncation signalled 405
- CHARACTER data type 91
- Character data types
  - Boolean comparisons 117
  - casting between 134
  - data strings 81
  - listed 118
- Character string
  - CHAR data type 90
  - CHARACTER VARYING data type 91
  - CLOB data type 91
  - converting to a DATE value 301
  - converting to a DATETIME value 302
  - DATETIME literals 93, 127, 167
  - INTERVAL literals 101
  - LVARCHAR data type 104
  - NCHAR data type 107
  - NVARCHAR data type 107
  - VARCHAR data type 117
  - with DELIMIDENT set 172
- CHARACTER VARYING data type
  - defined 91
- Character-based applications 195, 212
- Check constraints
  - creation-time value 156, 158
  - syschecks data 20
  - syscheckdrdep data 21
  - syscoldepend data 23
  - sysconstraints data 28
- Check constraints, definition of 381
- chkenv utility 141
  - error message 144
  - syntax 144
- Chunks 122
- Class libraries, shared 228

- CLIEN\_LABEL environment variable 150
- CLIENT\_LOCALE environment variable 157
- Client/server
  - DataBlade API 122
  - default database 193
  - INFORMIXSQLHOSTS environment variable 194
  - shared memory communication segments 193
  - stacksize for client session 195
- CLOB data type
  - casting unavailable 91
  - code-set conversion 92
  - collation 92
  - defined 91
  - inserting data 92
  - multibyte characters 92
  - syscolattns data 21
- CLOB TEXT 115
- CLOSE DATABASE statement, effect on database locks 436
- CLOSE statement 203
- Clustering 14, 39, 43
- CMCONFIG environment variable 149
- Code sets
  - East Asian 91, 167
  - EBCDIC 80
  - ISO 8859-1 34
- Collation 115
  - CHAR data type 90, 90
  - CLOB data type 92
  - GL\_COLLATE table 65
  - NCHAR data type 107
  - NVARCHAR data type 107
  - server\_attribute data 80
- Collection data type
  - casting matrix 138
  - defined 130
  - empty 130
  - LIST 103
  - MULTISET 106
  - SET 113
  - sysattrtypes data 17
  - sysxtddesc data 74
  - sysxtotypes data 74, 75
- COLLECTION data type
  - coltype code 23
- Collection data types
  - accessing 281, 286
  - counting elements in 303, 303
  - description of 286
  - element, searching for with IN 288
  - simple 286
  - updating 375, 375
  - using the CARDINALITY function 303
- collection delimiters 113, 130
- Collection subquery
  - description of 345
  - ITEM keyword 346, 347
  - using ITEM keyword in 347
- Collection types
  - in an SPL routine 462
  - in DELETE statement 361
- Collection values, inserting into columns 368
- Collection variable
  - defining, restrictions on 467
  - nested 286, 287
  - selecting 287
- collection-derived table
  - using in SPL 497
- Collection-derived table 348
  - accessing elements in a collection 348
  - description of 345, 493
  - restrictions on 348
- Collections, with INSERT statement 368
- Colon
  - cast ( :: ) operator 137
  - DATETIME delimiter 93
  - INTERVAL delimiter 101
- Color and intensity screen attributes 195
- Column number, using 242
- Column-level privileges
  - sysstabauth data 4
  - sysstabauth table 64
- Columns
  - changing data type 81, 133
  - constraints (sysconstraints) 28
  - default values (sysdefaults) 29
  - definition of 233
  - descending order 240
  - description of 227
  - hashed 39
  - in relational model 227
  - in superstores\_demo database 217
  - in trigger-event definitions 529
  - inserting BLOB data 87
  - label on 353
  - ordering the selection of 239
  - range of values 28
  - row-type, definition of 283
  - syscolumns data 23
- columns Information Schema view 76
- Combine function 13
- Comment indicator 141
- Comment lines 141
- COMMIT WORK statement
  - closing cursors 451
  - releasing locks 441, 451
  - setting SQLCODE 424
- Committed read 80
- Committed Read isolation level ( Informix )
  - 444
  - Communications support module 189
- commutator function
  - definition 460
- Commutator function 52
- Comparison condition, description of 247
- Compiling
  - ESQL/C programs 149
  - INFORMIXC setting 187
  - JAVA\_COMPILER setting 199
  - multithreaded ESQL/C applications 214
- Complex data type 129, 132
  - collection types 130
  - ROW types 132
  - sysattrtypes data 17
- Compliance
  - ANSI/ISO standard for SQL 76, 153
  - sql\_languages.conformance 79
  - X/Open CAE standards 76
  - XPG4 standard 78
- Composite index 41
- Compound query 350
- Concatenation ( || ) operator 139
- concsn.cfg file 189
- Concurrency
  - access modes 449
  - active set 415
  - ANSI isolation levels 444
  - Cursor Stability isolation ( Informix )
    - 445
    - database lock 436
    - deadlock 450
    - description of 394, 433
    - Informix isolation levels
      - 444
    - isolation level 442
    - kinds of locks 435
    - lock duration 441
    - lock scope 436
    - multiple programs 434
    - table lock 437
- Confidence level 37
- Configuration file
  - .cshrc file 142
  - .informix 141, 144, 174, 175
  - .login file 142
  - .profile file 142
  - for communications support module 189
  - for connectivity 187, 193, 194
  - for database servers 174, 201
  - for High-Performance Loader 205
  - for MaxConnect 187
  - for terminal I/O 195
- Configuration parameters
  - DBSPACETEMP 166
  - DEF\_TABLE\_LOCKMODE 175
  - DIRECTIVES 176
  - DISABLE\_B162428\_XA\_FIX 185
  - EXT\_DIRECTIVES 31, 177
  - ISOLATION\_LOCKS 445
  - MITRACE\_OFF 68, 69
  - OPCACHEMAX 192
  - OPT\_GOAL 203
  - OPTCOMPIND 202
  - RESIDENT 178
  - shared memory base 186
  - SQL\_LOGICAL\_CHAR 65
  - STACKSIZE 195
  - STMT\_CACHE 212
  - USEOSTIME 93
- CONNECT DEFAULT statement 193
- Connect privilege 9, 71
- CONNECT statement 163, 190, 193
- Connections
  - INFORMIXCONRETRY environment variable 189
  - INFORMIXCONTIME environment variable 190
  - INFORMIXSERVER environment variable 193
- Connectivity information 186, 194
- Constant expressions 364
- Constraints
  - check
    - creation-time value 158
    - syschecks data 20
    - syscheckudrdep data 21
    - syscoldepend data 23
  - column
    - sysconstraints data 28
  - not null
    - collection data types 106, 113, 130
  - NOT NULL
    - collection data types 103
    - syscoldepend data 23
    - syscolumns data 23
    - sysconstraints data 28



- object mode 47
- primary key
  - sysconstraints data 28
  - sysreferences data 56
  - unique SERIAL values 111
  - unique SERIAL8 values 112
- referential
  - sysconstraints data 28
  - sysreferences data 56
- table
  - sysconstraints data 28
- unique
  - sysconstraints data 28
  - sysviolations data 72
  - violations 72
- Constraints, entity integrity 381
- Constructors 113, 130
- Conversion function, description of 301
- Converting data types
  - DATE and DATETIME 136
  - INTEGER and DATE 136
  - number and string 136
  - number to number 135
  - retyping a column 133
- Coordinated deletes 425
- Coordinated Universal Time (UTC) 313
- Correlated subquery
  - definition of 335
  - restriction with cascading deletes 384
- COUNT function
  - and GROUP BY 321
  - as aggregate function 293
  - count rows to delete 360
  - use in a subquery 362
  - with DISTINCT 293
- CPFIRST environment variable 149
- CPU cost 210
- CREATE ACCESS\_METHOD statement 14
- CREATE CAST statement 20, 136
- CREATE DATABASE statement 163
  - setting shared lock 436
  - SQLWARN after 405
- CREATE DISTINCT TYPE statement 75, 98, 217
- CREATE EXTERNAL TABLE statement 34, 35
- CREATE FUNCTION FROM statement, in embedded languages 464
- CREATE FUNCTION statement 57
  - inside CREATE FUNCTION FROM statement 464
  - using 454
  - WITH LISTING IN clause 515
- CREATE FUNCTION, return clause 458
- CREATE IMPLICIT CAST statement 217
- CREATE INDEX statement 41, 43, 65, 172
  - storage options 172
- CREATE INDEX statement, locking table 437
- CREATE OPAQUE TYPE statement 108
- CREATE OPERATOR CLASS statement 48
- CREATE OPTICAL CLUSTER statement 48
- CREATE PROCEDURE FROM statement, in embedded languages 464
- CREATE PROCEDURE statement 57, 200
  - inside CREATE PROCEDURE FROM 464
  - using 454
  - WITH LISTING IN clause 515
- CREATE ROLE statement 56
- CREATE ROUTINE FROM statement 57, 200
- CREATE ROW TYPE statement 23, 108
- CREATE SCHEMA statement 4
- CREATE SEQUENCE statement 61
- CREATE SYNONYM statement 62, 63

- CREATE TABLE statement
  - assigning data types 81
  - cascading deletes 383
  - collection types 286
  - default lock mode 175
  - default privileges 200
  - hierarchy 289
  - multiset columns 346
  - ON DELETE CASCADE clause 359
  - primary keys 382
  - row type columns 281
  - SET constructor 113
  - setting the lock mode 440
  - smart large object columns 304
  - typed table 281
  - typed tables 108
- CREATE TEMP TABLE statement 166
- CREATE TRIGGER statement 70, 525
- CREATE VIEW statement 4, 72
- CREATE XADATASOURCE statement 73
- CREATE XADATASOURCETYPE statement 73
- Cross join 271
- Cross-database SQL operations 397
- Cross-server connection requirements 400
- Cross-server SQL operations 397
- Currency symbol 105, 162
- Current date 29, 154
- CURRENT function
  - as constant expression 364
  - comparing column values 297
  - using 297
- CURRENT keyword 123
- Cursor
  - active set of 414
  - closing 451
  - declaring 411
  - definition of 411
  - end of transaction 451
  - for insert 427
  - for update 431, 442
  - opening 412, 414
  - retrieving values with FETCH 412
  - scroll 413
  - sequence of program operations 411
  - sequential 413, 415
- Cursor Stability isolation level ( Informix )
  - 445
- Cyclic query 384

## D

- Data corruption 9, 21
- Data definition statements 420
- Data dependencies
  - syscheckudrdep data 21
  - syscoldepend data 23
  - sysdepend data 30
- Data dictionary 3
- Data distributions 9, 31, 171
- Data encryption functions 318
- Data integrity 79, 380
  - failures 391
- Data models, description of 220
- Data pages 21, 41, 65
- Data replication 394
- data type collation 115
- data type restrictions 115
- data type restrictions in Boolean expression 115
- data type UPDATE statements 115

- Data types
  - approximate 78
  - automatic conversions 408
  - BIGINT 86
  - BIGSERIAL 87
  - BLOB 87
  - BOOLEAN 88
  - BYTE 89
  - casting 133, 138
  - CHAR 90
  - CHARACTER 91
  - CHARACTER VARYING 91
  - classified by category 81
  - CLOB 91
  - collection 130
  - collection, accessing 281, 286
  - complex 129
  - conversion 133, 364, 408
  - DATE 92
  - DATETIME 93
  - DEC 96
  - DECIMAL 96
  - distinct 132
  - DISTINCT 98
  - DOUBLE PRECISION 99
  - exact numeric 78
  - extended 129
  - fixed point 97
  - FLOAT 99
  - floating-point 96, 99, 114
  - IDSSECURITYLABEL 100, 119
  - inheritance 108
  - INT 100
  - INT8 100
  - INTEGER 100
  - internal 81
  - INTERVAL 101
  - length (syscolumns) 27
  - LIST 103
  - LVARCHAR 104
  - MONEY 105
  - MULTISET 106
  - named ROW 108
  - NCHAR 107, 107
  - NUMERIC 107
  - NVARCHAR 107
  - opaque 132
  - OPAQUE 108
  - Opaque data types
    - smart large objects 122
  - REAL 108
  - ROW 108, 110
  - sequential integer 112
  - SERIAL 111
  - SERIAL8 112
  - SET 113
  - simple large object 122
  - SMALLFLOAT 114
  - SMALLINT 115
  - smart large object 122
  - summary list 81
  - unique numeric value 112
  - unnamed ROW 110
  - VARCHAR 117
- Data-type promotion 119
- Database identifiers 172
- Database object
  - constraints as a 384
  - index as a 384
  - object modes 384
  - trigger as a 384

- violation detection 384
- Database object mode
  - examples 385
- Database server administrator (DBSA) 4
- Database Server Administrator (DBSA) 195
- Database servers
  - archiving 393
  - attributes in Information Schema view 80
  - code set 80
  - default connection 193
  - default isolation level 80
  - identifying host computer name 313
  - identifying version number 313
  - locking tables 437
  - optimizing queries 203
  - pathname for 163
  - remote 174
  - role separation 195
  - server name 29, 163
  - signalled in SQLWARN 405
  - statement caching 452
- DATABASE statement 163
  - locking 436
  - SQLWARN after 405
- Databases
  - ANSI-compliant 231
  - compared to file 221
  - concurrent use 223
  - control of 223
  - data types 81
  - Databases
    - superstores\_demo 217
  - definition of 227
  - demonstration databases
    - superstores\_demo 217
  - external 396
  - identifiers 172
  - joins in stores\_demo 215
  - locking 436
  - management of 226
  - modifying contents of 223
  - object-relational 217
  - object-relational, description of 228
  - objects, sysobjstate data 47
  - privileges 71
  - relational, description of 226
  - remote 396
  - server 223
  - stores\_demo 215
  - superstores\_demo
    - demonstration database 217
  - syscrd 4
  - sysmaster 4
  - sysutils 4
  - sysuid 4
- DataBlade modules 228
  - Client and Server API 122
  - data types (sysbuiltintypes) 4
  - trace messages (sysracemsgs) 68, 69
  - user messages (syserrors) 34
- DATE data type
  - abbreviated year values 154
  - casting to integer 136
  - coltype code 23
  - converting to a character string 301
  - converting to DATETIME 136
  - defined 92
  - display format 157
  - functions returning 297
  - in expressions 123, 126
  - in ORDER BY sequence 240
- in relational expressions 247
- international date formats 92
- source data 126
- DATE function, as conversion function 301
- DATE() function 126, 157
- DATETIME data type
  - abbreviated year values 154
  - coltype code 23
  - converting to a character string 301
  - converting to DATE 136, 136
  - defined 93
  - display format 167, 167
  - displaying format 300
  - EXTEND function 126
  - extending precision 124
  - field qualifiers 93
  - functions returning 297
  - in expressions 123, 128
  - in ORDER BY sequence 240
  - in relational expressions 247
  - international formats 93, 93, 101
  - length (syscolumns) 27
  - literal values 93
  - localized values 93
  - precision and size 93
  - source data 127
  - two-digit year values and DBDATE variable 93
  - year to fraction example 93
- DATETIME values, formatting 300
- DAY function 297
- DAY keyword
  - DATETIME qualifier 93
  - INTERVAL qualifier 101
  - UNITS operator 92, 127
- DB-Access
  - creating database with 420
- DB-Access utility 9, 77, 145, 160, 163, 167, 193
- DBA privilege 34, 68, 69, 71
- DBA routines 52
- DBACCNOIGN environment variable 152, 153
- DBANSIWARN environment variable 153
- DBBLOBBUF environment variable 154, 154
- DBCENTURY environment variable
  - defined 154
  - effect on functionality of DBDATE 157
  - expanding abbreviated years 93, 155
- DBDATE environment variable 92, 93, 157, 364
- DBDELIMITER environment variable 159
- DBEDIT environment variable 159
- dbexport utility 159
- DBFLTMASK environment variable 160
- DBINFO function, in SELECT statement 313
- dbinfo utility 160
- DBINFO\_DBSPACE\_RETURN\_NULL\_FOR\_INVALID\_PARTITION environment variable 160
- DBLANG environment variable 161, 161
- dbload utility 87, 89, 115, 159
- DBMONEY environment variable 105, 162
- DBPATH environment variable 163
- DBPRINT environment variable 165
- DBREMOTECMD environment variable 165
- dbschema utility 52
- DBSECADM role 100, 119
- DBSERVERALIASES configuration parameter 400
- DBSERVERNAME configuration parameter 400
- DBSERVERNAME function, in INSERT statement 364
- DBSERVERNAME function, in SELECT statement 313
- dbservername.cmd batch file 147
- dbspace
  - for BYTE or TEXT values 19
  - for system catalog 4
  - for table fragments 36
  - for temporary tables 166
  - name 172
- dbspace, name returned by DBINFO function 313
- DBSPACETEMP configuration parameter 166
- DBSPACETEMP environment variable 166
- DBTEMP environment variable 167
- DBTIME environment variable 93, 167, 167
- DBUPSPACE environment variable 171
- Deadlock detection 450
- DEC data type 96
- DECIMAL data type
  - built-in casts 135, 136
  - coltype code 23
  - defined 96
  - disk storage 97
  - display format 160, 162
  - fixed point 97
  - floating point 96
  - length (syscolumns) 27
- DECIMAL data type, signalled in SQLWARN 405
- Decimal digits, display of 160
- Decimal point
  - DBFLTMASK setting 160
  - DBMONEY setting 162
  - DECIMAL radix 97
- Decimal separator 162
- DECLARE CURSOR statement 427
- DECLARE statement 203
  - description of 411
  - FOR INSERT clause 427
  - FOR UPDATE 431
  - SCROLL keyword 413
  - WITH HOLD clause 451
- DECODE function 314
- DECRYPT\_BINARY function 91, 318
- DECRYPT\_CHAR function 91, 318
- DEF\_TABLE\_LOCKMODE configuration parameter 175
- DEF\_TABLES\_LOCKMODE configuration parameter 440
- Default database locale 10
- Default values
  - in column 381
  - using 410
- DEFAULT\_ATTACH environment variable 172
- Defaults
  - C compiler 187
  - Century 154, 167
  - CHAR length 90
  - character set for SQL identifiers 172
  - compilation order 149
  - configuration file 201
  - connection 193
  - data type 110
  - database server 163, 193
  - DATE display format 92
  - DATE separator 157
  - DATETIME display format 93
  - DECIMAL precision 96
  - disk space for sorting 171
  - fetch buffer size 174
  - heap size 199
  - index storage location 172
  - isolation level 80

- join method 202
- level of parallelism 205
- lock mode 175
- message directory 161
- MONEY scale 105
- operator class 14, 48
- printing program 165
- query optimizer goal 203
- sysdefaults.default 29
- table privileges 200
- temporary dbspace 166
- terminfo directoty 213
- text editor 159
- DEFINE statement of SPL 111, 112
- defined Data types 115
- Delete MERGE operations 362
- Delete privilege 36, 64, 200
- DELETE statement 72
- DELETE statements 9
  - collection types 361
  - coordinated deletes 425
  - count of rows 423
  - description of 359
  - developing 362
  - duplicate rows 429
  - embedded 402, 423
  - lock mode 448
  - number of rows 404
  - preparing 418
  - remove all rows 359
  - row types 361
  - selected rows 360
  - specific rows 360
  - transactions with 424
  - using 423
  - using subquery 362
  - WHERE clause restriction 362
  - with cursor 425
  - with supertables 361
- Delete trigger 70
- Delete using TRUNCATE 359
- DELIMIDENT environment variable 172
- Delimited identifiers 172, 172
- Delimiter
  - for DATETIME values 93
  - for fields 159
  - for identifiers 172
  - for INTERVAL values 101
- demonstration databases
  - stores\_demo 215
- Demonstration databases
  - tables 217
- Descending index 41
- Descending order in SELECT 240
- DESCRIBE statement 185
- Describe-for-updates 185
- destroy() support function 129
- Detached index 172
- Deutsche mark (DM) currency symbol 162
- Diagnostics table 72
  - description of 388
  - example of privileges 390
  - examples of starting 388
- Difference set operation 356
- DIRECTIVES configuration parameter 176
- Directives for query optimization 176, 202, 203
- Dirty Read isolation level ( Informix )
  - 444
- Disabled database objects 72

- Disk space
  - for data distributions 171
  - for temporary data 166
- Display label
  - in ORDER BY clause 267
  - with SELECT 264
- Distinct data types
  - casts 138
  - sysxdtypes data 75
- DISTINCT data types
  - defined 98
  - sysxtddesc data 74
  - sysxdtypes data 75, 98
- DISTINCT keyword
  - relation to GROUP BY 321
  - using in SELECT 242
  - using with COUNT function 293
- Distinct-type variable 469
- Distributed Computing Environment (DCE) 214
- Distributed queries 129, 174
- Distributed SQL operations 397
- DOCUMENT clause, use in SPL routine 461
- Dollar (\$) sign 105, 162
- Domain of column 381
- Dot notation 284
- double (C) data type 99
- double hyphen (–) comment indicator 462
- Double-precision floating-point number 99
- DROP CAST statement 217
- DROP DATABASE statement 163
- DROP FUNCTION statement 52
- DROP INDEX statement 65
- DROP INDEX statement, locking table 437
- DROP OPTICAL CLUSTER statement 48
- DROP PROCEDURE statement 52
- DROP ROUTINE statement 52
- DROP ROW TYPE statement 108
- DROP SEQUENCE statement 215
- DROP TABLE statement 215
- DROP TYPE statement 98, 108
- DROP VIEW statement 77, 215
- Duplicate values, finding 268
- Dynamic routine-name specification
  - for SPL function 509
  - for SPL routine 509
  - rules for 510
- Dynamic SQL
  - description of 402, 417
  - freeing prepared statements 420

## E

- EBCDIC collation 80
- Editor, DBEDIT setting 159
- EMACS text editor 159
- Embedded SQL
  - definition of 401
  - languages available 401
- Empty set 130
- ENCRYPT\_AES function 318
- ENCRYPT\_DES function 91
- ENCRYPT\_TDES function 91, 318
- End of data
  - signal in SQLCODE 404, 409
  - signal only for SELECT 430
  - SQLCODE 412
  - when opening cursor 412
- Enterprise Replication 4
- Entity integrity 381
- env utility 144
- ENVIGNORE environment variable
  - defined 141, 174
  - relation to chkenv utility 144
- Environment configuration file
  - debugging with chkenv 144
  - setting environment variables in UNIX 141, 141
- Environment variables
  - ANSIOWNER 148
  - CLIENT\_LABEL 150
  - CLIENT\_LOCALE 157, 157
  - CMCONFIG 149
  - Colon
    - pathname separator 199
  - command-line utilities 146
  - CPFIRST 149
  - DBACCOIGN 152, 153
  - DBANSIWARN 153
  - DBBLOBBUF 154, 154
  - DBCENTURY 154
  - DBDATE 92, 93, 157
  - DBDELIMITER 159
  - DBEDIT 159
  - DBFLTMASK 160
  - DBINFO\_DBSpace\_RETURN\_NULL\_FOR\_INVALID\_PARTNUM 1
  - DBLANG 161
  - DBMONEY 105, 162
  - DBPATH 163
  - DBPRINT 165
  - DBREMOTECMD 165
  - DBSPACETEMP 166, 166
  - DBTEMP 167
  - DBTIME 93, 167
  - DBUPSPACE 171
  - DEFAULT\_ATTACH 172
  - DELIMIDENT 172
  - displaying current settings 144, 146
  - ENVIGNORE 174
  - FET\_BUF\_SIZE 174
  - GL\_DATE 92, 93, 156
  - GL\_DATETIME 93, 156
  - how to set
    - in Bourne shell 143
    - in C shell 143
    - in Korn shell 143
  - how to set in Bourne shell 143
  - how to set in Korn shell 143
  - IFMXMONGOAUTH 175
  - IFX\_DEF\_TABLE\_LOCKMODE 175
  - IFX\_DIRECTIVES 176
  - IFX\_EXTDIRECTIVES 31, 177
  - IFX\_LARGE\_PAGES 178
  - IFX\_LOB\_XFERSIZE 179
  - IFX\_LONGID 179
  - IFX\_NETBUF\_PVTPool\_SIZE 180
  - IFX\_NETBUF\_SIZE 180
  - IFX\_NO\_SECURITY\_CHECK 181
  - IFX\_NO\_TIMELIMIT\_WARNING 181
  - IFX\_NODBPROC 182
  - IFX\_NOT\_STRICT\_THOUS\_SEP 182
  - IFX\_ONTAPE\_FILE\_PREFIX 182
  - IFX\_PAD\_VARCHAR 182
  - IFX\_SMX\_TIMEOUT 183, 184
  - IFX\_SMX\_TIMEOUT\_RETRY 184
  - IFX\_UNLOAD\_EILSEQ\_MODE 184
  - IFX\_UPDESC 185
  - IFX\_XASTDCOMPLIANCE\_XAEND 185
  - IFX\_XFER\_SHMBase 186
  - IMCADMIN 186
  - IMCONFIG 187
  - IMCSERVER 187
  - INF\_ROLE\_SEP 195, 195
  - INFORMIXC 187

INFORMIXCMCONUNITNAME 188  
 INFORMIXCMNAME 188  
 INFORMIXCONCSMCFG 189  
 INFORMIXCONRETRY 189  
 INFORMIXCONTIME 190  
 INFORMIXCPPMAP 192  
 INFORMIXDIR 192  
 INFORMIXOPCACHE 192  
 INFORMIXSERVER 193  
 INFORMIXSHMBASE 193  
 INFORMIXSQLHOSTS 194, 194  
 INFORMIXSTACKSIZE 195  
 INFORMIXTERM 195  
 INTERACTIVE\_DESKTOP\_OFF 196  
 ISM\_COMPRESSION 197  
 ISM\_DEBUG\_FILE 197  
 ISM\_DEBUG\_LEVEL 197  
 ISM\_ENCRYPTION 197  
 ISM\_MAXLOGSIZE 198  
 ISM\_MAXLOGVERS 198  
 JAR\_TEMP\_PATH 198  
 JAVA\_COMPILER 199  
 JVM\_MAX\_HEAP\_SIZE 199  
 LD\_LIBRARY\_PATH 199  
 LIBPATH 200  
 limitations 140  
 manipulating in Windows environments 145  
 modifying settings 143  
 NODEFDAC 200  
 ONCONFIG 201  
 ONINIT\_STDOUT 201  
 OPT\_GOAL 203  
 OPTCOMPIND 202  
 OPTMSG 202  
 OPTOFC 203  
 overriding a setting 141, 174  
 PATH 204, 204  
 Pathname  
     for client or shared libraries 199  
 PDQPRIORITY 204  
 PLCONFIG 205  
 PLOAD\_LO\_PATH 206  
 PLOAD\_SHMBASE 206  
 PSM\_ACT\_LOG 206, 207  
 PSM\_DBS\_POOL 207  
 PSM\_DEBUG 207  
 PSM\_DEBUG\_LOG 208  
 PSM\_LOG\_POOL 208  
 PSORT\_DBTEMP 209  
 PSORT\_NPROCS 209  
 RTREE\_COST\_ADJUST\_VALUE 210  
 rules of precedence in UNIX 145  
 rules of precedence in Windows 148  
 scope of reference 146  
 setting 146  
     at the command line 141  
     in a configuration file 141  
     in a login file 141  
     in a shell file 142  
     in Windows environments 145  
     with the System applet 146  
 setting in autoexec.bat 146  
 SHLIB\_PATH 211  
 SRV\_FET\_BUF\_SIZE 211  
 standard UNIX system 140  
 STMT\_CACHE 212  
 TERM 212  
 TERMCAP 213  
 TERMINFO 213  
 THREADLIB 214  
     types of 140  
     unsetting 143, 146, 172  
     USE\_DTENV 93, 93  
     USETABLENAME 215  
     view current setting 144  
     where to set 142  
 equal() support function 129  
 Equality (=) operator 91  
 Equals (=) relational operator 248, 271  
 Equi-join 271, 271  
 Era-based dates 167  
 Error checking  
     simulating errors 520  
     SPL routines 518, 521  
 Error message files 161, 406  
 Error messages  
     applying fixed 535  
     for trigger failure 535  
     generating a variable 536  
     generating in a trigger 535  
     retrieving trigger text in a program 535, 536  
 Errors  
     after DELETE 424  
     codes for 404  
     dealing with 409  
     detected on opening cursor 412  
     during updates 391  
     inserting with a cursor 428  
     ISAM error code 404  
 ESCAPE keyword, using in WHERE clause 258  
 esql command 149, 187  
 ESQL/C  
     cursor use 411, 416  
     DATETIME routines 167  
     DELETE statement in 423  
     delimiting host variables 402  
     dynamic embedding 402, 417  
     error handling 409  
     esqlc command 149  
     fetching rows from cursor 412  
     host variable 402, 403  
     indicator variable 409  
     INSERT in 427  
     long identifiers 179  
     message chaining 202  
     multithreaded applications 214  
     overview 400, 423, 423  
     preprocessor 401  
     program compilation order 149  
     scroll cursor 413  
     selecting single rows 407  
     SQL Communications Area 403  
     SQLCODE 403  
     SQLERRD fields 404  
     static embedding 401  
     UPDATE in 431  
 Exact numeric data types 78  
 EXCLUSIVE keyword  
     in DATABASE statement 436  
 Exclusive lock 435  
 Executable programs 204  
 EXECUTE FUNCTION statement  
     with SPL 505  
 EXECUTE IMMEDIATE statement, description of 420  
 Execute privilege 50, 200  
     DBA keyword, effect of 514  
     objects referenced by a routine 513  
 EXECUTE PROCEDURE statement  
     with SPL 505  
 EXISTS keyword 355  
     in a WHERE clause 339  
     in SELECT statement 342  
 explain output file 171  
 Explicit cast 20, 137  
 Explicit pathnames 146, 164  
 Explicit temporary tables 166  
 Exponent 97  
 Exponential notation 96  
 export utility 143  
 export\_binary() support function 129  
 export() support function 129  
 Expression  
     CASE 265  
     date-oriented 297  
     description of 262  
     display label for 264  
     in SPL routine 475  
 Expression-based fragmentation 37, 39, 156, 158  
 EXT\_DIRECTIVES configuration parameter 31, 177  
 EXTEND function 126  
     using in expression 300  
     with DATE and DATETIME values 297  
 EXTEND role 511  
 Extended data types 75, 129, 129, 217  
 Extensibility, description of 228  
 Extensible Markup Language (XML) 91  
 Extension checking (DBANSIWARN) 153  
 Extents, changing size 9  
 External database 63, 396  
 External directives for query optimization 177  
 External routines 52, 511  
 External tables  
     sysextcols data 34  
     sysextdfiles data 35  
     sysexternal data 35  
     systables data 65  
 External view 63  
 extspace 14  
**F**  
 FALSE setting  
     BOOLEAN value 88  
 Farsi locales 90  
 FET\_BUF\_SIZE environment variable 174  
 Fetch buffer 174  
 Fetch buffer size 174  
 FETCH statement 203, 413  
     ABSOLUTE keyword 413  
     description of 412  
     sequential 413  
     with sequential cursor 415  
 Field delimiter  
     DBDELIMITER 159  
     Statements of SQL  
         LOAD 159  
         UNLOAD 159  
     Utilities  
         dbexport 159  
 Field of a ROW data type 132  
 Field projection 284  
 Field qualifier  
     DATETIME values 93  
     EXTEND function 126  
     INTERVAL values 101  
 Field, definition of 283  
 Fields of a ROW data type 132  
 File extensions  
     .a 179  
     .cfg 189

- .cmd 147
- .ec 149
- .ecp 149
- .iem 161
- .jar 198
- .rc 141, 145, 174, 175
- .so 179
- .sql 77, 163, 163, 172
- .std 201, 212
- Files
  - environment configuration files 144
  - installation directory 192
  - permission settings 141
  - shell 142
  - temporary 166, 167, 209
  - temporary for SE 167
  - termcap, terminfo 195, 213, 213
- Files, compared to database 220
- FILETOBLOB function 87
- FILETOCLOB function 91
- Filtering mode 47, 72, 385
- Finalization function 13
- FIRST clause
  - description of 259
  - using 260
  - with ORDER BY clause 260
- Fixed point decimal 97, 105, 162
- Fixed-length opaque data types 23
- Fixed-length UDT 75
- FLOAT data type
  - built-in casts 135, 136
  - coltype code 23
  - defined 99
  - display format 160, 162
- Floating-point decimal 96, 99, 114, 160
- FLUSH statement
  - count of rows inserted 428
  - rollback 428
  - writing rows to buffer 427
- FOR UPDATE keywords
  - conflicts with ORDER BY 425
  - not needed in ANSI-compliant database 432
  - specific columns 432
- FOREACH statement 476
- Foreign key 382
- Formatting
  - DATE values with DBDATE 157
  - DATE values with GL\_DATE 167
  - DATETIME values with DBTIME 167
  - DATETIME values with GL\_DATETIME 167
  - DATETIME values with USE\_DTENV 167
  - DECIMAL(p) values with DBFLTMASK 160
  - FLOAT values with DBFLTMASK 160
  - MONEY values with DBMONEY 162
  - SMALLFLOAT values with DBFLTMASK 160
- Formatting mask
  - with DBDATE 157
  - with DBFLTMASK 160
  - with DBMONEY 162
  - with DBTIME 167
  - with GL\_DATE 167
  - with GL\_DATETIME 167
  - with USE\_DTENV 167
- FRACTION keyword
  - DATETIME qualifier 93
- FRAGMENT BY clause 166
- Fragment-level statistics 37
- Fragmentation
  - distribution strategy 39
  - encrypted distribution 37

- expression 37, 39, 156, 158
- fragment statistics 37
- list 39
- PDQPRIORITY environment variable 205
- PSORT\_NPROCS environment variable 210
- round robin 37, 39
- setting priority levels for PDQ 204
- sysfragauth data 36
- sysfragdist data 37
- sysfragments data 39
- FREE statement, freeing prepared statements 420
- FROM clause
  - subqueries in 338
- FROM keyword 9, 20
- FROM keyword, alias names 277
- Function keys 195
- Functional index 41, 131, 172
- Functions
  - aggregate 293
  - aggregates of arithmetic expressions 296
  - conversion 301
  - DATE 301
  - date-oriented 297
  - DBINFO 313
  - DECODE 314
  - for BLOB columns 87
  - for CLOB columns 91
  - for MULTISSET columns 106
  - in SELECT statements 292
  - INITCAP 306
  - LOWER 305
  - LPAD 309
  - name confusion in SPL routine 472
  - NVL 316
  - REPLACE 307
  - RPAD 310
  - smart large object 304
  - string manipulation 305
  - SUBSTR 309
  - SUBSTRING 308
  - support for complex types 129
  - time 297
  - TO\_CHAR 301
  - TO\_DATE 302
  - UPPER 306
- Functions, data encryption 318
- fwritable gcc option 187

**G**

- gcc compiler 187
- Generic B-trees 41
- GET DIAGNOSTICS statement 34, 407
- getenv utility 141
- GETHINT function 318
- GL\_COLLATE table 65
- GL\_CTYPE table 65
- GL\_DATE environment variable 92, 93, 156, 157
- GL\_DATETIME environment variable 93, 156
- Global network buffer pool 180
- Global variable
  - declaring 472
  - description of 472
- GLS environment variables 145
- GNU C compiler 187
- GRANT statement 56, 64
- GRANT statement, in embedded SQL 421, 421
- GRANT USAGE ON LANGUAGE statement 511
- Granularity, of locks 436
- Graphic characters 195

- Greater than or equal to (>=) relational operator 249
- GROUP BY clause 89, 115, 166
  - description of 320
- GROUP BY keywords
  - column number with 321
  - description of 320
- GROUP BY TEXT 115
- Group informix 161

## H

- Hash-join 202
- hash() support function 129
- Hashed columns 39
- Hashing parameters 63
- HAVING clause, description of 320
- HAVING keyword 323
- HCL
- Informix
- ESQL/C
  - 149, 157, 167, 179, 202
- Heap size 199
- Hebrew locales 90
- HEX function, using in expression 313
- Hexadecimal digits 159
- hierarchy
  - table and row 289
- HIGH INTEG keywords
  - ALTER TABLE statement 122
  - CREATE TABLE statement 122
- HIGH keyword
  - in UPDATE STATISTICS statement 37
  - PDQPRIORITY 204
  - UPDATE STATISTICS 9, 31
- High-Performance Loader 205
- Histogram 31
- Hold cursor, definition of 451
- Host language 79
- Host variable 87, 89, 115, 131, 403
  - delimiter for 402
  - description of 402
  - fetching data into 412
  - in DELETE statement 423
  - in INSERT statement 427
  - in UPDATE statement 431
  - in WHERE clause 407
  - INTO keyword sets 407
  - null indicator 409
  - restrictions in prepared statement 418
  - truncation signalled 405
- HOURL keyword
  - DATETIME qualifier 93
  - INTERVAL qualifier 101
- HP-UX operating system 211
- HTML (Hypertext Markup Language) 91
- Hyphen
  - DATETIME delimiter 93
  - INTERVAL delimiter 101

**I**

- I/O overhead 210
- IBM
- Informix
- Storage Manager (ISM)
  - 198
- IDSSECURITYLABEL data type
  - definition 100
- IF statement, in SPL 478
- IFXMONGOAUTH environment variable 175
- IFX\_DEF\_TABLE\_LOCKMODE environment variable 175, 440
- IFX\_DIRECTIVES environment variable 176

IFX\_EXTDIRECTIVES environment variable 31, 177  
 IFX\_EXTEND\_ROLE configuration parameter 511  
 IFX\_LARGE\_PAGES environment variable 178  
 IFX\_LOB\_XFERSIZE environment variable 179  
 IFX\_LONGID environment variable 179  
 IFX\_NETBUF\_PVTPOOL\_SIZE environment variable 180  
 IFX\_NETBUF\_SIZE environment variable 180  
 IFX\_NO\_SECURITY\_CHECK environment variable 181  
 IFX\_NO\_TIMELIMIT\_WARNING environment variable 181  
 IFX\_NODBPROC environment variable 182  
 IFX\_NOT\_STRICT\_THOUS\_SEP environment variable 182  
 IFX\_ONTAPE\_FILE\_PREFIX environment variable 182  
 IFX\_PAD\_VARCHAR environment variable 182  
 IFX\_SMX\_TIMEOUT environment variable 183, 184  
 IFX\_SMX\_TIMEOUT\_RETRY environment variable 184  
 IFX\_UNLOAD\_EILSEQ\_MODE environment variable 184  
 IFX\_UPDESC environment variable 185  
 IFX\_XASTDCOMPLIANCE\_XAEND environment variable 185  
 IFX\_XFER\_SHMBASE environment variable 186  
 imcadmin administrative tool 186  
 IMCADMIN environment variable 186  
 IMCCONFIG environment variable 187  
 IMCSERVER environment variable 187  
 IMPEXP data type 137  
 IMPEXPBIN data type 137  
 Implicit cast 20, 137  
 Implicit connection 193  
 Implicit temporary tables 166  
 import\_binary() support function 129  
 import() support function 129  
 IN clause 166  
 IN keyword 89, 106, 111, 113, 139  
   to form an intersection 355  
   using in WHERE clause 247  
 IN relational operator 339  
 in SET VIOLATIONS TABLE statement  
   MAX ROWS keywords 388  
   USING keyword 388  
 IN TABLE keywords  
   CREATE INDEX statement 438  
 IN TABLE storage option 172  
 Index  
   attached 39, 156, 172, 210  
   B-tree 41, 172  
   clustered 41, 43  
   composite 41, 41  
   default values for attached 210  
   descending 41  
   detached 172  
   distribution scheme 172  
   forest of trees 172  
   fragmented 37, 39  
   functional 41, 131, 172  
   nonfragmented 172, 172  
   of data types 81  
   of system catalog tables 10  
   R-Tree 172  
   sysindexes data 41  
   sysindices data 43  
   sysobjstate data 47  
   threads for sorting 210  
   unique 28, 41, 111, 112  
 Index key structure 43  
 Index privilege 64  
 Indicator variable, definition of 409  
 Indirect typing 111, 112  
 Industry standards, compliance with 79  
 INF\_ROLE\_SEP environment variable 195, 195  
 Information Schema views  
   accessing 77  
   columns 78  
   defined 76, 76  
   generating 77  
   server\_info 80  
   sql\_languages 79  
   tables 78  
 Informational messages 34  
 Informix  
   database, object-relational databases 228  
   Informix extension checking (DBANSIWARN) 153  
   informix owner name 9, 20, 31, 41, 43, 65, 161, 195  
   informix.rc file 141, 145, 175  
   INFORMIXC environment variable 187  
   INFORMIXCMCONUNITNAME environment variable 188  
   INFORMIXCMNAME environment variable 188  
   INFORMIXCONCSMCFG environment variable 189  
   INFORMIXCONRETRY environment variable 189  
   INFORMIXCONTIME environment variable 190  
   INFORMIXCPPMAP environment variable 192  
   INFORMIXDIR environment variable 192  
   INFORMIXOPCACHE environment variable 192  
   INFORMIXSERVER environment variable 193  
   INFORMIXSHMBASE environment variable 193  
   INFORMIXSTACKSIZE environment variable 195  
   INFORMIXTERM environment variable 195  
   Inheritance hierarchy 46, 110  
   INITCAP function, as string manipulation function 306  
   Initialization function 13, 57  
   Input support function 104  
   input() support function 129  
   Insert cursor  
     definition of 427  
     using 429  
   Insert MERGE operations 364  
   Insert privilege 36, 64, 200  
   INSERT statements 68, 72, 93, 130, 152, 157  
     and end of data 430  
     collection columns 368  
     constant data with 429  
     count of rows inserted 428  
     description 363  
     embedded 427  
     inserting  
       collections 368  
       into supertables 368  
       multiple rows 370  
     lock mode 448  
     named row type 366  
     null values in collection 369  
     number of rows 404  
     SELECT restrictions 370  
   SELECT statement in 370  
   selected columns 365  
   serial values 365  
   smart large objects in 369  
   unnamed row type 367  
   VALUES clause 363  
   with row-type columns 366  
   with SELECT statement 370  
   Insert trigger 70  
   Inserting rows of constant data 429  
   Installation directory 192  
   INSTEAD OF trigger 70, 532  
   INT data type 100  
   INT8 data type  
     built-in casts 135, 136  
     coltype code 23  
     defined 100  
     using with SERIAL8 87  
   INTEG keyword 122  
   INTEGER data type  
     built-in casts 135, 136  
     coltype code 23  
     defined 100  
     length (syscolumns) 27  
   Intensity attributes 195  
   Intent lock 448  
   INTERACTIVE\_DESKTOP\_OFF environment variable 196  
   Internationalized trace messages 69  
   Interprocess communications (IPC) 193  
   Intersection  
     definition of 355  
     set operation 354  
   INTERVAL data type  
     coltype code 23  
     defined 101  
     field delimiters 101  
     in expressions 123, 123, 128, 129  
     in relational expressions 247  
     length (syscolumns) 27  
   INTO clause 413  
   INTO keyword  
     choice of location 413  
     in FETCH statement 413  
     mismatch signalled in SQLWARN 405  
     restrictions in INSERT 370  
     restrictions in prepared statement 418  
     retrieving multiple rows 411  
     retrieving single rows 407  
   INTO TEMP keywords, description of 279  
   ipcshm protocol 193  
   IPCSTR connection 400  
   IS NOT NULL keywords 252  
   IS NULL keywords 252  
   IS NULL operator 89  
   ISAM error code 404  
   ISM\_COMPRESSION environment variable 197  
   ISM\_DEBUG\_FILE environment variable 197  
   ISM\_DEBUG\_LEVEL environment variable 197  
   ISM\_ENCRYPTION environment variable 197  
   ISM\_MAXLOGSIZE environment variable 198  
   ISM\_MAXLOGVERS environment variable 198  
   ISO 8859-1 code set 80  
   Isolation level 80, 202  
     ANSI 444, 446  
     Cursor Stability ( Informix )  
       445  
     description of 442  
     dirty read 444

- Informix
  - 444
  - read uncommitted 444
  - repeatable read 446
- ISOLATION\_LOCKS configuration parameter 445
- ITEM keyword, collection subquery 346, 347
- Iterator functions 13

## J

- Japanese eras 167
- Jar management procedures 198
- JAR\_TEMP\_PATH environment variable 198
- Java virtual machine (JVM) 148, 198, 199, 199
- JAVA\_COMPILER environment variable 199
- JIT compiler 199
- Join
  - ANSI outer-join syntax 329
  - associative 274
  - composite 268
  - condition 268
  - creating 270
  - cross 271
  - definition of 236, 268
  - equi-join 271
  - in MERGE statement 362, 364, 371
  - Informix
    - outer join syntax 329
    - left outer 330
    - multiple-table join 275
    - natural 274
    - nested simple 332
    - on derived tables 330
    - outer 328
    - right outer 331
    - self-join 325
    - simple 268
  - Join methods 202
  - Join operations 9, 166
  - JVM\_MAX\_HEAP\_SIZE environment variable 199

## K

- KEEP ACCESS TIME keywords
  - ALTER TABLE statement 122
  - CREATE TABLE statement 122
- Key
  - primary 28, 56, 56, 72, 217
- Key scan 14
- Keyboard I/O
  - INFORMIXTERM setting 195
  - TERM setting 212
  - TERMCAP setting 213
  - TERMINFO setting 213
- keyword MATCHES 115
- Keywords
  - in a subquery 339
  - in a WHERE clause 247
- Korn shell 141, 142

## L

- Label 264, 353
- Label-based access control (LBAC) 100, 119
- Language
  - C 57, 149, 187
  - C++ 192
  - CLIENT\_LOCALE setting 157
  - DBLANG setting 161
  - Extensible Markup Language (XML) 91
  - Hypertext Markup Language (HTML) 91
  - Informix

- ESQL/C
  - 122, 131, 214
  - Java 148, 198, 199
  - sql\_languages information schema view 79
  - Stored Procedure Language (SPL) 131, 156, 158
  - syslangauth data 46
  - sysroutinelangs data 57
- Large pages for virtual memory segments 178
- Large-object data type
  - defined 121
  - listed 118
- LD\_LIBRARY\_PATH environment variable 199
- Leaf pages 39
- Left outer join 330
- LENGTH function
  - on TEXT or BYTE strings 311
  - on VARCHAR 311
  - use in expression 311
- Less than or equal to (>=) relational operator 249
- LET statement 473, 473
- libos.a library 179
- LIBPATH environment variable 200
- LIKE 115
- LIKE clause
  - in SPL function 470
- LIKE keyword
  - description of 254
  - using in WHERE clause 247
- LIKE keyword of SPL 111, 112
- LIKE operator 89, 139
- Linearized code 69
- List
  - of data types 81
  - of system catalog tables 10
- LIST data type
  - coltype code 23
- LIST data type, defined 103
- LO\_handles() support function 129
- LOAD statement 87, 89, 115, 159
- Local variable, description of 466
- Locales
  - collation order 65
  - multibyte 91
  - of trace messages 69
  - right-to-left 90
- Localized collation 119
- LOCK TABLE statement, locking a table explicitly 437
- Lock-table overflow 175
- Locking
  - and concurrency 394
  - behavior of different lock types 448
  - deadlock 450
  - description of 435
  - end of transaction 451
  - integrity 434
  - intent locks 448
  - lock duration 441
  - number of rows to lock 445
  - row and key locks 438
  - scope of lock 436
  - setting lock mode 449
  - time limit 450
  - types of locks 435
    - coarse index lock 441
    - database lock 436
    - exclusive 435
    - page lock 438, 439
    - promotable 435

- promotable lock 442
- row and key locks 438
- shared 435
- smart-large-object locks 441
- table lock 437
- update cursor 442
- update lock 447
- WAIT keyword 449
  - with DELETE 424
- LOCKMODE keyword 175
- LOCOPY function 87, 91
- LOG keyword
  - ALTER TABLE statement 122
  - CREATE TABLE statement 122
- Logging mode 21
- Logical characters 119
- Logical log
  - and backups 393
  - description of 392
- Logical operator
  - = (equals) 253
  - AND 253
  - NOT 253
  - OR 253
- Long identifiers
  - client version 179
  - IFX\_LONGID setting 179
  - Information Schema views 77
- Loop, exiting with RAISE exception 521
- LOTOFILE function 87, 91
- LOW keyword
  - PDQPRIORITY 204
  - UPDATE STATISTICS 31
- LOWER function, as string manipulation function 305
- Lowercase mode codes 52
- Lowercase privilege codes 4, 22, 64
- LPAD function, as string manipulation function 309
- LVARCHAR data type
  - casting opaque types 137
  - coltype code (for client) 23
  - defined 104

## M

- Machine notes 195
- Machine-independent integer types 27
- Magnetic storage media 19
- Mantissa precision 78, 97
- Map file for C++ programs 192
- MATCHES 115
- MATCHES keyword
  - using in WHERE clause 247
- MATCHES operator 89, 139
- MATCHES relational operator
  - in WHERE clause 254
- MAX function, as aggregate function 294
- MaxConnect 186, 187
- MEDIUM keyword 9, 28, 31
- MEDIUM keyword, in UPDATE STATISTICS statement 37
- Membership operator 139
- Memory cache, for staging blobspace 192
- MERGE statement 72
  - using Insert join 364
  - using Update join 371
- MERGE statements
  - using Delete join 362
- Message file
  - specifying subdirectory with DBLANG 161
- Messages

- chaining 202
- error in syserrors 34
- optimized transfers 202
- reducing requests 203
- trace message template 69
- warning in syserrors 34
- mi\_collection\_card() function 103, 106, 113
- mi\_db\_error\_raise() function 34
- Microsoft C compiler 187
- MIN function, as aggregate function 294
- MINUTE keyword
  - DATEIME qualifier 93
  - INTERVAL qualifier 101
- MITRACE\_OFF configuration parameter 68, 69
- mkdir utility 161
- MODE ANSI keywords, specifying transactions 393
- MODERATE INTEG keywords
  - ALTER TABLE statement 122
  - CREATE TABLE statement 122
- Modifiers
  - CLASS 52
  - COSTFUNC 52
  - HANDLESNULLS 52
  - INTERNAL 52
  - NEGATOR 52
  - NOT VARIANT 52
  - PARALLELIZABLE 52
  - SELCONST 52
  - STACK 52
  - VARIANT 52
- MODIFY NEXT SIZE keywords 9
- MONEY data type
  - built-in casts 136
  - coltype code 23
  - defined 105
  - display format 162
  - in INSERT statement 364
  - international money formats 105
  - length (syscolumns) 27
- MONTH function
  - using, TIME function MONTH 298
- MONTH function, as time function 297
- MONTH keyword
  - DATEIME qualifier 93
  - INTERVAL qualifier 101
- Multibyte characters
  - CLOB data type 92
- Multiple-table join 275
- Multiple-Table SELECTs 268
- MULTISET data type
  - coltype code 23
  - constructor 130
  - defined 106
- MULTISET keyword
  - collection subquery 346
- Multithreaded application, definition of 401

## N

- N setting
  - sysroleauth.is\_grantable 56
- Named ROW data type
  - casting permitted 138
  - defined 108
  - defining 108
  - equivalence 108
  - inheritance 46, 108
  - typed tables 108
- Named row type, in VALUES clause 366
- Namer ROW data type

- coltype code 23
- National Language Support (NLS) 119
- Natural join 274
- NCHAR data type
  - collation order 107
  - coltype code 23
  - defined 107
  - multibyte characters 107
- NCHAR data type, querying on 233
- Negator functions 52
- Nested dot notation 131
- Nested ordering, in SELECT 241
- Nested-loop join 202
- Network buffers 180
- Network environment variable, DBPATH 163
- NFS directory 167
- NLS data types
  - in system catalog tables 10
- NO KEEP ACCESS TIME keywords
  - ALTER TABLE statement 122
  - CREATE TABLE statement 122
- no setting of NODEFDAC 200
- NODEFDAC environment variable 200
- NODEFDAC environment variable, effect on privileges of public 512
- NOLOG keyword
  - ALTER TABLE statement 122
  - CREATE TABLE statement 122
- Non-default database locales 10
- NONE setting
  - JAVA\_COMPILER 199
- Nonfragmented index 172
- Nonprintable characters
  - CHAR data type 91
  - TEXT data type 116
  - VARCHAR data type 117
- NOT BETWEEN keywords in WHERE clause 250
- Not equal (!=) relational operator 249
- NOT EXISTS keywords 356
- NOT IN keywords 356
- NOT logical operator 253
- NOT NULL 115
- NOT NULL constraint
  - collection elements 103, 106, 113, 130
  - syscoldepend data 23
  - sysconstraints data 28
- NOT NULL keywords 89, 103
- NOT operator 139
- NOT VARIANT routine 52
- NULL data type
  - coltype code 23
- NULL value
  - allowed or not allowed 13, 23
  - BOOLEAN literal 88
  - BYTE data type 89
- Null values
  - detecting in ESQL 409
  - testing for 252
  - with logical operator 253
- Numeric data types
  - casting between 135
  - casting to character types 136
  - listed 118
- NVARCHAR data type
  - collation order 107
  - coltype code 23
  - defined 107
  - multibyte characters 107
- NVARCHAR data type, querying on 233
- NVL function 316

## O

- Object mode
  - description of 384
  - disabled 385
  - enabled 385
  - filtering 385
- Object mode of database objects 47
- Object-relational database, description of 228
- Object-relational schema 217
- ODBC driver 199, 211
- OFF setting
  - IFX\_DIRECTIVES 176, 177
  - PDQPRIORITY 204
- ON DELETE CASCADE option 383
- ON EXCEPTION statement
  - scope of control 519
  - trapping errors 518
  - user-generated errors 520
- ON setting
  - IFX\_DIRECTIVES 176, 177
- ONCONFIG environment variable 201
- onconfig.std file 212
- oninit command 175
- ONINIT\_STDOUT environment variable 201
- ONLINE keyword
  - CREATE INDEX statement 438
  - DROP INDEX statement 438
- Online transaction processing (OLTP) 39
- onload utility 87, 89, 115, 393
- onpload utility 206
- onsecurity utility 181
- onstat utility 140
- onunload utility 393
- Opaque data types
  - cast matrix 138
  - comparing 137
  - storage 104
  - sysxtddesc data 74
  - sysxtdtypes data 75
- OPAQUE data types
  - defined 108
- Opaque-type variable 469
- OPCACHEMAX configuration parameter 192
- OPEN statement 203, 412
- Opening a cursor 414
- Operator class
  - sysams data 14
  - sysindices data 43
  - sysopclasses data 48
- operator LIKE 115
- Operator precedence 139
- operator TEXT 115
- OPT\_GOAL configuration parameter 203
- OPT\_GOAL environment variable 203
- OPTCOMPIND configuration parameter 202
- OPTCOMPIND environment variable 202
- Optical cluster
  - INFORMIXOPCACHE setting 192
  - sysblobs.type column 19
  - sysopclstr data 48
- Optimizer
  - setting IFX\_DIRECTIVES 176
  - setting IFX\_EXTDIRECTIVES 177
  - setting OPT\_GOAL 203
  - setting OPTCOMPIND 202
  - setting OPTOFC 203
- Optimizer directives
  - sysdirectives data 31
- OPTMSG environment variable 202
- OPTOFC environment variable 203



- OR logical operator 253
- OR operator 139
- OR relational operator 251
- ORDER 115
- ORDER BY clause 89, 166
- ORDER BY keywords
  - ascending order 240
  - DESC keyword 240, 242
  - display label with 267
  - multiple columns 241
  - relation to GROUP BY 321
  - restrictions in INSERT 370
  - restrictions with FOR UPDATE 425
  - select columns by number 242
  - sorting rows 239
- Ordinal positions 103
- Outer-join syntax
  - ANSI 329
  - Informix 329
- Output support function 104
- output() support function 129
- Overflow error 97
- Owner routines 52, 200

**P**

- Page footers in sbspaces 122
- Page headers in sbspaces 122
- PAGE lock mode 65, 175
- Page locking 438
- Parallel distributed queries, setting with PDQPRIORITY 204
- Parallel sorting, setting with PSORT\_NPROCS 209
- Partial characters 90
- Partial-column index 43
- Parts explosion 416
- PATH environment variable 204, 204
- Pathname
  - Configuration file
    - for terminal I/O 213
  - for C compiler 187, 187
  - for C++ map file 192
  - for conccm.cfg file 189
  - for connectivity information 194
  - for database server 163
  - for dynamic-link libraries 200, 211
  - for environment-configuration file 144
  - for executable programs 204
  - for installation 192
  - for message files 161, 161
  - for parallel sorting 209
  - for remote shell 165
  - for smart-large-object handles 206
  - for temporary .jar files 198
  - for termcap file 213
  - for terminfo directory 213
  - separator symbols 204
- PDQ
  - OPTCOMPIND environment variable 202
  - PDQPRIORITY environment variable 204
- Percentage ( %) symbol 167
- Performance
  - effect of concurrency 434
  - increasing with stored routines 454
- Period
  - DATE delimiter 157
  - DATETIME delimiter 93
  - INTERVAL delimiter 101
- Permissions 141, 161
- PLCONFIG environment variable 205

- plconfig file 205
- PLOAD\_LO\_PATH environment variable 206
- PLOAD\_SHMBASE environment variable 206
- PostScript 91
- Precedence rules
  - for casts 137
  - for lock mode 175
  - for SQL operators 139
  - for UNIX environment variables 145
  - for Windows environment variables 148
- Precision
  - of currency values 105
  - of numbers 78, 96, 99, 100, 100, 114
  - of time values 93, 101, 124, 128
- PREPARE statement 65
  - description of 418
  - error return in SQLERRD 404
  - multiple SQL statements 418
- Prepared statement 65
- Primary access method 14, 63
- Primary key 28, 56, 72, 111, 112, 217
- Primary key constraint, definition of 382
- Primary key, definition of 381
- Primary thread 195
- printenv utility 144
- Printing with DBPRINT 165
- Private environment-configuration file 144, 174
- Private network buffer pool 180, 180
- Private synonym 65
- Privilege
  - default table privileges 200
  - on columns (syscolauth table) 22
  - on procedures and functions (sysprocauth table) 50
  - on table fragments (sysfragauth table) 36
  - on tables (systabauth table) 64
  - on the database (sysusers table) 71
  - on UDTs and named row types (sysxtdtypeauth) 74
- Privileges
  - database-level 378
  - displaying 379
  - needed to modify data 378
  - on a database 378
  - overview 223
  - table-level 379
- Procedure-type variables 470
- Program variables
  - SPL 402
- Projection, definition of 235
- Projects, description of 228
- Promotable lock 435, 442
- Protected routines 52
- Protected rows 100, 119
- Pseudo-machine code (p-code) 51
- PSM\_ACT\_LOG environment variable 206
- PSM\_CATALOG\_PATH environment variable 207
- PSM\_DBS\_POOL environment variable 207
- PSM\_DEBUG environment variable 207
- PSM\_DEBUG\_LOG environment variable 208
- PSM\_LOG\_POOL environment variable 208
- PSORT\_DBTEMP environment variable 209
- PSORT\_NPROCS environment variable 209
- Public synonym 63, 65
- public user name 77
- Purpose functions 14
- PUT statement
  - constant data with 429
  - count of rows inserted 428
  - insert data 427

- sends returned data to buffer 427
- status code 428
- putenv utility 141

## Q

- Qualifier field
  - DATETIME 93
  - EXTEND 127
  - INTERVAL 101
  - UNITS 127
- Qualifier, existential 342
- Query
  - audit 342
  - compound 350
  - cyclic 384
  - self-referencing 384
  - stated in terms of data model 222
- Query optimizer
  - directives 176, 177
  - sysdistrib data 31
  - sysproplan data 55
  - updating distribution data 9
- Quoted string
  - DATE and DATETIME literals 127
  - DELIMIT setting 172
  - INTERVAL literals 101
  - invalid with BYTE 89
  - LVARCHAR data type 104
- Quoted string invalid with TEXT 115
- Quoted string, as constant expression 364

## R

- R-tree index 172, 210
- RAISE EXCEPTION statement 518
- RANGE function, as aggregate function 294
- Re-entrant trigger, description of 532
- Read committed 80
- Read Committed isolation level (ANSI) 444
- Read uncommitted 80
- Read Uncommitted isolation level (ANSI) 444
- Recursive relationship, example of 416
- recv() support function 129
- REFERENCES keyword
  - in CREATE FUNCTION statement 456
  - in CREATE PROCEDURE statement 456
- REFERENCES keyword, in SPL function 467
- References privilege 22, 64
- Referential constraint 28, 56, 72
- Referential constraint, definition of 382
- Referential integrity, definition of 382
- Relational database, description of 226
- Relational model
  - join 236
  - projection 234
  - selection 234
- Relational operation 234
- Relational operators 91, 139
  - BETWEEN 249
  - EXISTS 339
  - IN 339
  - in a WHERE clause 247
  - LIKE 254
  - MATCHES 254
  - NULL 252
  - OR 251
- Remote database 396
- Remote database server 63, 174
- Remote shell 165
- Remote tape devices 165
- RENAME SEQUENCE statement 215
- Repeatable read 202
- Repeatable read isolation level 446

- REPLACE function, as string manipulation function 307
- Replica identifier 39
- Replication
  - of data 394
  - transparency 394
- RESIDENT configuration parameter 178
- Resource contention 204
- Resource Grant Manager (RGM) 39
- Resource privilege 9
  - Role
    - sysusers data 71
  - System catalog
    - authorization identifiers 71
- Return types, in SPL function 458
- REVOKE statement 64
- REVOKE statement, in embedded SQL 421, 421
- Right outer join 331
- Right-to-left locales 90
- Role
  - default role 71
  - INF\_ROLE\_SEP setting 195
  - sysroleauth data 56
- Role separation 195
- Roles
  - default 223
  - definition 223
- ROLLBACK WORK statement
  - closes cursors 451
  - releases locks 441, 451
  - setting SQLCODE 424
- Rolling-window fragmentation 39
- Round-robin fragmentation 37, 39
- Routines
  - DataBlade API routine 68
  - DATETIME formatting 167
  - identifier 52
  - owner 52
  - privileges 50
  - protected 52
  - restricted 52
  - Stored Procedure Language (SPL) 131
  - syserrors data 34
  - syslangauth data 46
  - sysprocauth data 50
  - sysprocbody data 51
  - sysprocedures data 52
  - sysprocplan data 55
  - sysroutinelangs data 57
  - systraceclasses data 68
  - sysracemsgs data 69
  - trigger 52
- ROW data types 132
  - casting permitted 138
  - dot notation with 284
  - equivalence 108
  - field projection 284
  - field projections in SELECT 285
  - field, definition of 283
  - fields 17, 132
  - in DELETE statement 361
  - inheritance 46, 108
  - inserting values 111
  - named 108, 132
  - selecting columns from 283
  - selecting data from 281
  - sysattrtypes data 17
  - sysxtddesc data 74
  - sysxtotypes data 74, 75
  - unnamed 110, 132

- updating 374
  - using asterisk notation with SELECT 285
- ROW lock mode 65, 175
- Row type columns
  - definition of 283
  - Null values 375
- Row-type data, selecting columns of 283
- Row-type variables, declaring 468
- ROWID, using to locate internal row numbers 268
- ROWIDS 14
- Rows
  - checking rows processed in SPL routines 521
  - definition of 227, 233
  - finding number of rows processed 313
  - in relational model 227
  - inserting 363
  - locking 438
  - number of rows returned 259
  - removing 359
  - updating 371
- RPAD function, as string manipulation function 310
- RTNPARAMTYPES data type 52
- RTREE\_COST\_ADJUST\_VALUE environment variable 210
- Runtime
  - warnings (DBANSIWARN) 153

**S**

- Sample size 31
- Sampling data 37
- SAVE EXTERNAL DIRECTIVES statement 177
- SBSAPACENAME configuration parameter 31, 37
- sbspaces
  - defined 91, 122
  - name 172
  - sysams data 14
  - syscolattribs data 21
  - sysstabamdata data 63
- Scale of numbers 78, 97, 160
- Scan cost 14
- Schema Tools 145
- Screens, example 535
- Scroll cursors
  - active set 415
  - definition of 413
- SCROLL keyword, using in DECLARE 413
- SECOND keyword
  - DATETIME qualifier 93
  - FRACTION keyword
  - INTERVAL qualifier 101
  - INTERVAL qualifier 101
- Secondary-access methods 14, 28, 43, 48, 108
- Security policy 100
- Select cursor
  - opening 412
  - using 411
- SELECT INTO TEMP statement 166
- Select list
  - display label 264
  - expressions in 262
  - functions in 292, 313
  - labels in 353
  - selecting all columns 238
  - selecting specific columns 242
  - specifying a substring in 246
- Select privilege 22, 64, 77, 200
- SELECT statements 9, 31

- accessing collections 281, 286
- active set 247, 407
- advanced 320
- aggregate functions in 293, 301
- alias names 277
- ALL keyword 339
- and end-of-data return code 430
- ANY keyword 339
- basic concepts 233
- collection expressions 345
- collection subquery 346
- collection-derived table 348
- compound query 350
- cursor for 411, 411
- date-oriented functions in 297
- description of 232
- display label 264
- DISTINCT keyword 242
- embedded 407, 409
- executing triggered actions 530
- EXISTS keyword 342, 342
- FIRST clause 259
- for joined tables 279
- for single tables 238, 313
- forms of 233
- functions 292, 313
- GROUP BY clause 321
- HAVING clause 323
- in UPDATE statement 372, 372
- INTO clause with ESQL 407
- INTO TEMP clause 279
- isolation level 442
- join 270
- multiple-table 268
- natural join 274
- ORDER BY clause 239
- outer join 328
- select list 234
- selecting a row type 281
- selecting a substring 246
- selecting expressions 262
- selection list 238
- self-join 325
- set operations 350
- simple 232
- single-table 238
- singleton 247, 407
- smart-large-object functions in 304
- stand-alone 530
- subquery 335
- UNION operator 350
- using
  - for join 236
  - for projection 235
  - for selection 234
  - using functions 292
- Select trigger, description of 529
- SELECT triggers 70
- Select, description of 228
- Selection, description of 234
- Selectivity constant 52, 52
- Self-join 4, 325
  - assigning column names with INTO TEMP 325
  - description of 325
- Self-referencing query 325, 384
- Semantic integrity 381
- send() support function 129
- SENDRECV data type 137
- Sequence
  - definition of 228

- syssequences data 61
- sys synonyms data 62
- sysstable data 63
- systabauth data 64
- stables data 65
- Sequential cursor, definition of 413
- Sequential integers
  - am\_id code 14
  - classid code 68
  - constrid code 28
  - extended\_id code 75
  - langid code 57
  - msgid code 69
  - opclassid code 48
  - planid code 55
  - procid code 52, 52
  - seqid code 61
  - SERIAL data type 111
  - SERIAL8 data type 112
  - tabid code 4, 61, 65
- SERIAL data type
  - coltype code 23
  - defined 111
  - generated number in SQLERRD 404
  - inserting a starting value 365
  - inserting values 111
  - last SERIAL value inserted 313
  - length (syscolumns) 27
  - resetting values 111
- SERIAL8 data type
  - assigning a starting value 113
  - coltype code 23
  - defined 112
  - inserting values 113
  - last SERIAL8 value inserted 313
  - length (syscolumns) 27
  - resetting values 113
  - using with INT8 87
- Serializable transactions 80
- server\_info Information Schema view 76
- Session ID, returned by DBINFO function 313
- SET clause, in UPDATE statement 373
- SET data type
  - coltype code 23
- SET data type, defined 113
- SET Database Object Mode statement 388
- SET ENVIRONMENT IFX\_AUTO\_REPREPARE statement 65
- SET ENVIRONMENT statement 141, 145, 202
- Set intersection 355, 529
- SET ISOLATION statement
  - and SET TRANSACTION 443, 443
  - use of 442
- SET keyword, in MERGE statement 371
- SET keyword, in UPDATE statement 372
- SET LOCK MODE statement, description of 449
- Set operation
  - difference 356
  - intersection 354
  - union 350
  - use of 350
- SET OPTIMIZATION statement 203, 203
- SET PDQPRIORITY statement 204
- SET SESSION AUTHORIZATION statement 52
- SET STMT\_CACHE statement 212, 212
- SET TRANSACTION statement
  - and SET ISOLATION 443
  - use of 442
- set utility 146
- setenv utility 143
- Setnet32 148
- Setnet32 utility 145
- Setting environment variables
  - in UNIX 141
  - in Windows 145
- SGML (Standard Graphic Markup Language) 91
- Shared class libraries 228
- Shared environment-configuration file 144
- Shared libraries 179
- Shared lock 435
- Shared memory
  - INFORMIXSHMBASE 193
  - PLOAD\_SHMBASE 206
- Shell
  - remote 165
  - search path 204
  - setting environment variables in a file 142
  - specifying with DBREMOTECMD 165
- SHLIB\_PATH environment variable 211
- simple large object
  - defined 89
- Simple large objects
  - defined 122
  - location (sysblobs) 19
- Simple large objects, SPL variable 467
- Single-precision floating-point number 108, 114
- Singleton SELECT statement 247, 407
- SITENAME function, in INSERT statement 364
- SITENAME function, in SELECT statement 313
- SMALLFLOAT data type
  - built-in casts 135, 136
  - coltype code 23
  - defined 114
  - display format 160, 162
- SMALLINT data type
  - built-in casts 135, 136
  - coltype code 23
  - defined 115
  - length (syscolumns) 27
- Smart large objects
  - defined 122
  - functions for copying 304
  - importing and exporting 304, 369
  - in an UPDATE statement 377
  - SPL variables 467
  - syscolattrs data 21
  - using SQL functions
    - in a SELECT statement 304
    - in an INSERT statement 369
- Smart-large-object handles 206
- Solaris operating system 178
- SOME keyword, beginning a subquery 339
- SOME operator 139
- Sort-merge join 202
- Sorting
  - DBSPACETEMP environment variable 166
  - nested 241
  - PSORT\_DBTEMP environment variable 209
  - PSORT\_NPROCS environment variable 209
  - with ORDER BY 240
- Space
  - DATETIME delimiter 93
  - INTERVAL delimiter 101
- Spatial queries 210
- Special character, protecting 258
- Specific name, for SPL routine 456
- SPL
  - assigning values to variables 473, 473, 475
  - FOREACH loop 476
- LET statement 473
  - parameter list 456
  - program variable 402
  - relation to SQL 453, 453
  - return clause 458
  - statement block 475
  - tracing triggered actions 533
  - using cursors 476
  - WITH LISTING IN clause 461
- SPL function
  - CALL statement 507
  - collection query 493
  - dynamic routine-name specification 509
  - large object variables 467
  - variant vs. nonvariant 460
  - WITH clause 460
- SPL routines 52, 131, 156, 158
  - adding comments to 461
  - as triggered action 528
  - collection data types 487
  - comments 462
  - compiler messages 515
  - CONTINUE statement 482
  - debugging 516
  - definition of 453
  - dot notation 486
  - dropping 464
  - dynamic routine-name specification 509
  - example of 463
  - exceptions 518, 521
  - EXECUTE PROCEDURE 528
  - executing 505
  - EXIT statement 482
  - exiting a loop 482
  - finding errors 515
  - FOR loop 481
  - IF..ELIF..ELSE structure 478
  - in an embedded language 464
  - in SELECT statements 316
  - introduction to 453
  - name confusion with SQL functions 471
  - passing data 528
  - privileges 510
  - return types 458
  - returning values 483
  - row-type data 486
  - specific name 456
  - SQL expressions 475
  - syntax error 515
  - system catalog entries 516
  - text of 516
  - TRACE statement 533
  - updating nontriggering columns 529
  - uses 454
  - variables, scope of 466
  - WHILE loop 481
  - writing 454
- SPL variables 131
- SQL
  - application languages 401
  - Application Programming Interfaces 401
  - compliance of statements with ANSI standard 230
  - cursor 411
  - description of 229
  - difference between Informix syntax and ANSI standard 230
  - dynamic statements 402
  - error handling 409

- history 230
- Informix
- SQL and ANSI SQL
  - 230
  - interactive use 230
  - standardization 230
  - static embedding 401
- SQL (Structured Query Language) 153
- SQL character set 172
- SQL Communications Area 153
  - altered by end of transaction 424
  - description of 403
  - inserting rows 428
- SQL statement cache 452
- sql\_languages Information Schema view 76
- SQL\_LOGICAL\_CHAR configuration parameter 65, 65, 119
- SQLCODE
  - end of data 412
  - negative values 410
- SQLCODE field
  - after opening cursor 412
  - and FLUSH operation 428
  - description of 403
  - end of data on SELECT only 430
  - end of data signalled 409
  - set by DELETE statement 423
  - set by PUT statement 428
- SQLERRD array
  - count of deleted rows 423
  - count of inserted rows 428
  - count of rows 430
  - description of 404
  - syntax of naming 403
- SQLERRM character string 406
- sqlhosts file 186, 193, 194, 400
- SQLHOSTS subkey 194
- SQLSTATE values 34, 407
- SQLSTATE variable
  - in non-ANSI-compliant databases 409
  - using with a cursor 412
- SQLSTATE, problem values 410
- sqltypes.h file 23
- SQLWARN array 153
  - description of 405
  - syntax of naming 403
  - with PREPARE 418
- SRV\_FET\_BUF\_SIZE environment variable 211
- Stack size 52, 195
- STACKSIZE configuration parameter 195
- Staging-area blobspace 192
- Standard deviation, aggregate function 295
- Standard Graphic Markup Language (SGML) 91
- START DATABASE statement 163
- START VIOLATIONS TABLE 388
- START VIOLATIONS TABLE statement 72
- STAT data type 31
- STATCHANGE configuration parameter 31, 37
- STATCHANGE table attribute 31, 37
- Statement block 475
- Statement cache 212
- Statement cache, SQL 452
- Statements of SQL
  - ALTER INDEX 43
  - ALTER OPTICAL CLUSTER 48
  - ALTER SEQUENCE 61, 215
  - ALTER TABLE 9, 56, 65, 215
  - CLOSE 203
  - CONNECT 163, 163, 190, 193
  - CREATE ACCESS\_METHOD 14
  - CREATE AGGREGATE 13
  - CREATE CAST 20, 136
  - CREATE DATABASE 163
  - CREATE DISTINCT TYPE 75, 98, 217
  - CREATE EXTERNAL TABLE 34, 35
  - CREATE FUNCTION 57, 200
  - CREATE IMPLICIT CAST 217
  - CREATE INDEX 4, 41, 43, 65, 172, 172
  - CREATE OPAQUE TYPE 75, 108
  - CREATE OPERATOR CLASS 48
  - CREATE OPTICAL CLUSTER 48, 48
  - CREATE PROCEDURE 51, 57
  - CREATE ROLE 56, 71
  - CREATE ROUTINE FROM 57
  - CREATE ROW TYPE 75, 108
  - CREATE SCHEMA AUTHORIZATION 4
  - CREATE SEQUENCE 61
  - CREATE SYNONYM 63
  - CREATE TABLE 29, 56, 63
  - CREATE TRIGGER 70
  - CREATE VIEW 72
  - CREATE XADATASOURCE 73
  - CREATE XADATASOURCETYPE 73
  - DATABASE 163
  - DECLARE 203
  - DELETE 9, 55, 72, 72
  - DESCRIBE 185
  - DROP CAST 217
  - DROP DATABASE 163
  - DROP FUNCTION 52
  - DROP INDEX 65
  - DROP OPTICAL CLUSTER 48
  - DROP PROCEDURE 52
  - DROP ROUTINE 52
  - DROP ROW TYPE 108
  - DROP SEQUENCE 215
  - DROP TABLE 215
  - DROP TYPE 98, 108
  - DROP VIEW 77, 215
  - FETCH 203
  - GET DIAGNOSTICS 34
  - GRANT 36, 56, 64, 64, 77
  - INSERT 72, 130, 152, 157
  - LOAD 89, 153, 153
  - MERGE 72
  - OPEN 203
  - PREPARE 65
  - RENAME SEQUENCE 215
  - RENAME TABLE 215
  - REVOKE 64, 71
  - SELECT 9, 31, 55, 166
  - SET ENVIRONMENT 202
  - SET OPTIMIZATION 203
  - SET PDQPRIORITY 204
  - SET SESSION AUTHORIZATION 52
  - SET STMT\_CACHE 212
  - START DATABASE 163
  - START VIOLATIONS TABLE 72
  - UNLOAD 154
  - UPDATE 152
  - UPDATE STATISTICS 9, 43, 171
  - UPDATE STATISTICS FOR PROCEDURE 55
  - UPDATE STATISTICS FOR TABLE 28
- Statements of SQL LOAD 115
- Statements of SQL UPDATE 115
- static option of ESQ/C 179
- Static SQL 401
- STATLEVEL table attribute 37
- STDEV function, as aggregate function 295
- STMT\_CACHE configuration parameter 212
- STMT\_CACHE environment variable 212
- STMT\_CACHE keyword 212
- Storage identifiers 172
- Stored procedure language (SPL) 52, 131, 156
- Stored routine, general programming 231
- stores\_demo database 215
  - join columns 215
- Stream pipe connection 400
- strings option of gcc 187
- Structured Query Language (SQL) 153
- Subquery
  - ALL keyword 339
  - ANY keyword 339
  - correlated 335, 341, 384
  - in DELETE statement 362
  - in FROM clause 338
  - in select list 337
  - in SELECT statement 335
  - in UPDATE statement 372
    - with SET clause 372
  - in WHERE clause 339
  - single-valued 340
- Subscripting
  - in a WHERE clause 258
  - SPL variables 470
- Subscripts 89
- Subscripts ( [] ), 115
- SUBSTR function, as string manipulation function 309
- Substring 246, 470
- SUBSTRING function 9
- SUBSTRING function, as string manipulation function 308
- Subtable 37, 39, 46, 219
- Subtype 46, 108
- SUM function, as aggregate function 296
- Summary
  - of data types 81
  - of system catalog tables 10
- superstores\_demo database
  - structure of tables 217
- Supertable 46, 219, 291
  - in a table hierarchy 289
  - inserting into 368
  - selecting from 291
  - using an alias 291
- Supertype 46, 108
- Support functions
  - DISTINCT data types 132
  - OPAQUE data types 108, 129
  - routine identifier 52
- Symbol table 52, 52
- Synonym
  - sys synonyms data 62
  - sysstable data 63
  - sysables data 65
  - USETABLENAME setting 215
- sysaggregates system catalog table 13
- sysams system catalog table 14
- sysattrtypes system catalog table 17
- sysautolocate system catalog table 18
- sysblobs system catalog table 19
- sysbuiltintypes table 4
- syscasts system catalog table 20, 133
- syschecks system catalog table 20
- syscheckudrdep system catalog table 21
- syscolattribs system catalog table 21
- syscolauth system catalog table 22
- syscoldepend system catalog table 23
- syscolumns system catalog table 23
- sysconstraints system catalog table 28
- syscrd database 4

- SYSDATE function, as time function 297, 364
- sysdbclose
  - disabling with IFX\_NODBPROC 182
- sysdbopen
  - disabling with IFX\_NODBPROC 182
- sysdefaults system catalog table 29
- sysdepend system catalog table 30
- sysdirectives system catalog table 31
- sysdistrib system catalog table 31
- sysdomains system catalog view 33
- syserrors system catalog table 34
- sysextcols system catalog table 34
- sysextdfiles system catalog table 35
- sysexternal system catalog table 35
- sysfragauth system catalog table 36
- sysfragdist system catalog table 37
- sysfragments system catalog table 39
- sysindexes system catalog table 41
- sysindexes system catalog tables 43
- sysinherits system catalog table 46
- syslangauth system catalog table 46
- syslogmap system catalog table 47
- sysmaster database 4
  - contrasted with system catalog tables 4
  - initialization 140
- sysobjstate system catalog table 47
- sysopclasses system catalog table 48
- sysopclstr system catalog table 48
- sysprocauth system catalog table 50
- sysprocbody system catalog table 51
- sysprocbody, system catalog table 516
- sysproccolumns system catalog table 52
- sysprocedures system catalog table 52
- sysprocplan system catalog table 55
- sysreferences system catalog table 56
- sysroleauth system catalog table 56
- sysroutinelangs system catalog table 57
- sysseclabelauth system catalog table 57
- sysseclabelcomponentelements system catalog table 58
- sysseclabelcomponents system catalog table 58
- sysseclabelnames system catalog table 59
- sysseclabels system catalog table 59
- syssecpolicycomponents system catalog table 60
- syssecpolicyexemptions system catalog table 60
- syssequences system catalog table 61
- sys surrogateauth system catalog table 61
- sys synonyms system catalog table 62
- sys syntable system catalog table 63
- sys tabamdata system catalog table 63
- sys tabauth system catalog table 64
- sys tables system catalog table 65
- System administrator (DBA) 4
- System applet 146
- System catalog
  - access methods 14, 63
  - access privileges 22, 36
  - accessing 9
  - altering contents 9
  - casts 20
  - columns 23
  - complex data types 17, 75
  - constraint violations 72
  - constraints 20, 23, 28
  - data distributions 31
  - database tables 65
  - default values 29

- defined 3
- dependencies 30
- discretionary access privileges 64
- drvurity policies 59
- example 4
- external directives 31
- external tables 34, 35, 35
- fragment distributions 37
- fragment privileges 36
- fragments 39
- indexes 41, 43
- inheritance 46
- list of tables 10
- messages 34
- operator classes 48
- optical clusters 48
- privileges 71, 74
- programming languages 46, 57
- referential constraints 28, 56, 72
- roles 56
- routine parameters 52
- routines 50, 52, 55
- security label components 58
- sequence objects 61
- simple large objects 19
- smart large objects 21
- synonyms 62
- text of routines 51
- trace classes 68
- trace messages 69
- triggers 69, 70
- updating 9
- use by database server 4
- user-defined aggregates 13
- user-defined data types 74, 75
- views 65, 72
- XA data source types 73
- XA data sources 73
- System catalog tables
  - synonyms 63
  - sys aggregates 13
  - sysams 14
  - sysattrtypes 17
  - sysautolocate 18
  - sysblobs 19
  - syscasts 20
  - syschecks 20
  - syscheckudrdep 21
  - syscolattris 21
  - syscolauth 22
  - syscoldepend 23
  - syscolumns 23
  - sysconstraints 28
  - sysdefaults 29
  - sysdepend 30
  - sysdirectives 31
  - sysdistrib 31
  - sysdomains 33
  - syserrors 34
  - sysextcols 34
  - sysextdfiles 35
  - sysexternal 35
  - sysfragauth 36
  - sysfragdist 37
  - sysfragments 39
  - sysindexes 41
  - sysindices 43
  - sysinherits 46
  - syslangauth 46
  - syslogmap 47
  - sysobjstate 47

- sysopclasses 48
- sysopclstr 48
- sysprocauth 50
- sysprocbody 51
- sysproccolumns 52
- sysprocedures 52
- sysprocplan 55
- sysreferences 56
- sysroleauth 56
- sysroutinelangs 57
- sysseclabelauth 57
- sysseclabelcomponentelements 58
- sysseclabelcomponents 58
- sysseclabelnames 59
- sysseclabels 59
- syssecpolicycomponents 60
- syssecpolicyexemptions 60
- syssequences 61
- sys surrogateauth 61
- sys synonyms 62
- sys syntable 63
- sys tabamdata 63
- sys tabauth 64
- sys tables 65
- sys traceclasses 68
- sys tracemsgs 69
- sys trigbody 69
- sys triggers 70
- sys users 71
- sys views 72
- sys violations 72
- sys xadatasources 73
- sys xasourcetypes 73
- sys xtddesc 74
- sys xtdtypeauth 74
- sys xtdtypes 75
- System catalogs
  - privileges in 379
  - querying 379
  - sysprocbody 516
  - sys tabauth 379
- System descriptor area 419
- SYSTEM() command, on NT 196
- sys traceclasses system catalog table 68
- sys tracemsgs system catalog table 69
- sys trigbody system catalog table 69
- sys triggers system catalog table 70
- sys users system catalog table 71
- sysutils database 4
- sysuid database 4
- sys views system catalog table 72
- sys violations system catalog table 72
- sys xadatasources system catalog table 73
- sys xasourcetypes system catalog table 73
- sys xtddesc system catalog table 74
- sys xtdtypeauth system catalog table 74
- sys xtdtypes system catalog table 75, 108, 108

## T

- tabid 4, 65
- Table
  - changing a column data type 133
  - dependencies, in sysdepend 30
  - description of 227
  - diagnostic 72
  - extent size 65
  - fragmented 37, 39
  - hashing parameters 63
  - hierarchy 37, 39, 46, 108, 219, 289
  - in relational model 227

- inheritance, sysinherits data 46
- loading data
  - with onload utility 393
- lock 437
- lock mode 65, 175
- nonfragmented 172
- not in the current database 253
- operations on a 228
- separate from large object storage 121
- structure in superstores\_demo database 217
- synonyms in sysstable 62
- sysstable data 65
- system catalog tables 13
- temporary 166, 167
- temporary in SE 167
- untyped, and unnamed ROW 110
- version value 65
- violations 72
- Table hierarchy
  - triggers in 529
  - UPDATE statements 376
- Table-based fragmentation 39
- Table-level privileges
  - PUBLIC 77
  - sysfragauth data 36
  - sysstabauth data 4, 64
- tables Information Schema view 76
- Tape management
  - setting DBREMOTECMD 165
- TCP/IP connection 400
- Temporary dbspace 166
- Temporary files 167
  - in SE, specifying directory with DBTEMP 167
  - setting DBSPACETEMP 166
  - setting PSORT\_DBTEMP 209
- Temporary tables 166
  - and active set of cursor 414
  - assigning column names 325
  - example 370
  - in SE, specifying directory with DBTEMP 167
  - specifying dbspace with DBSPACETEMP 166
- TERM environment variable 212
- TERMCAP environment variable 213
- termcap file
  - setting INFORMIXTERM 195
  - setting TERMCAP 213
- Terminal handling
  - setting INFORMIXTERM 195
  - setting TERM 212
  - setting TERMCAP 213
  - setting TERMINFO 213
- terminfo directory 195, 213
- TERMINFO environment variable 213
- TEXT 115
- TEXT argument 115
- TEXT Character string TEXT 115
- TEXT data type 115, 115
  - coltype code 23
  - increasing buffer size 154
  - length (syscolumns) 27
  - nonprintable characters 116
  - restrictions
    - with GROUP BY 321
  - setting buffer size 154
  - sysblobs data 19
  - sysfragments data 39
  - using LENGTH function on 311
    - with control characters 116
    - with relational expressions 247
- TEXT data type IS NULL 115
- TEXT data type restrictions 115
- Text editor 159
- thousands separator 182
- Thousands separator 105
- thread flag of ESQL/C 214
- THREADLIB environment variable 214
- Time data types
  - arithmetic 123
  - length (syscolumns) 27
  - listed 118
- Time function
  - description of 297
  - use in SELECT 292
- TIME function
  - DAY and CURRENT 297
  - WEEKDAY 299
  - YEAR 300
- Time values
  - DBCENTURY setting 154
  - DBDATE setting 157
  - DBTIME setting 167
  - GL\_DATETIME settings 167
  - USEOSTIME configuration parameter 93
- Time-limited licenses
  - (IFX\_NO\_TIMELIMIT\_WARNING) 181
- Timezone
  - setting TZ 214
- TO keyword
  - DATETIME qualifier 93
  - EXTEND function 126
  - INTERVAL qualifier 101
- TO\_CHAR function, as conversion function 301
- TO\_DATE function, as conversion function 302
- TODAY function, in constant expression 312, 364
- TODAY operator 29
- Trace class 68
- Trace messages 69
- TRACE statement
  - debugging an SPL routine 516
  - output 534
- Trace statements 69
- Transaction isolation level 80, 202
- Transaction logging 21, 80
  - contents of log 393
  - description of 392
- Transactions
  - description of 391
  - end of 451
  - example with DELETE 424
  - locks held to end of 442
  - locks released at end of 441
  - logging 392
  - use signalled in SQLWARN 405
- Trigger action
  - definition of 524
  - REFERENCING clause 526
- Trigger event
  - definition of 524
  - example of 524
- Trigger events
  - recommendation against for auditing 529
- Trigger routines 52, 529
- Triggered action
  - BEFORE and AFTER 525
  - FOR EACH ROW 526
  - generating an error message 535
- in relation to triggering statement 524
- SELECT statements 530
- statements 522
- tracing 533
- using 525
- using SPL routines 528
- WHEN condition 527
- Triggers
  - creating 523
  - creation-time value 156, 158
  - declaring the name 524
  - definition of 522
  - in a table hierarchy 529
  - INSTEAD OF 532
  - re-entrant 532
  - restrictions on Select trigger execution 531
  - select
    - defining on a table hierarchy 532
  - Select 529
  - sysobjstate data 47
  - systrigbody data 69, 69
  - systriggers data 70
  - when to use 523
- TRUE setting
  - BOOLEAN values 88
  - sysams table 14, 14, 14, 14, 14
- TRUNCATE statement 359
- Truncation 90
- Truncation, signalled in SQLWARN 405
- TYPE keyword 110
- Typed table
  - definition of 281
  - inserting rows 365
  - selecting from 282
- TZ environment variable 214

## U

- UDT indexes 210
- Unary arithmetic operators 139
- Uncommitted read 80
- Under privilege 64
- UNION keyword, in set operations 350
- UNION operator, display labels with 353
- Union set operation 350
- Unique constraint 72, 111, 112
- Unique index 41, 111
- Unique keys 14
- UNIQUE keyword, in SELECT statement 242
- Unique numeric values
  - SERIAL data type 111
  - SERIAL8 data type 112
- UNITS operator 92, 123, 127, 139
- UNIX
  - BSD, default print utility 165
  - environment variables 140
  - PATH environment variable 204
  - System V
    - default print utility 165
    - terminfo libraries 195, 213
    - temporary files 209
  - TERM environment variable 212
  - TERMCAP environment variable 213
  - TERMINFO environment variable 213
- UNLOAD statement 154, 159
- Unnamed ROW data type
  - coltype code 23
  - declaring 110
  - defined 110
  - inserting values 111
- Unnamed row type, in VALUES clause 367
- unset utility 143

- unsetenv utility 143
- Unsetting an environment variable 143
- untyped 467
- Untyped table 65
- Update cursor 446
- Update cursor, definition of 431
- UPDATE keyword 432
- Update locks, retaining 447
- Update MERGE operations 371
- Update privilege 22, 36, 64, 200
- UPDATE statement 72
- UPDATE statements 185
  - and end of data 430
  - collection data types 375
  - description of 371, 371
  - embedded 431
  - failures 391
  - lock mode 448
  - number of rows 404
  - preparing 418
  - restrictions on subqueries 373
  - SET clause 373
  - smart large objects 377
  - WHERE clause 372
  - with a supertable 376
  - with row data types 374
  - with uniform values 372
- UPDATE STATISTICS FOR PROCEDURE statement 55
- UPDATE STATISTICS statement 43, 171
  - and DBUPSPACE environment variable 171
  - effect on sysdistrib table 31
  - sysindices (index statistics) 48
  - sysindices data 43
  - updating system catalog tables 9
- Update trigger 70
- UPPER function, as string manipulation function 306
- Uppercase mode codes 52
- Uppercase privilege codes 4, 22, 64
- USE\_DTEENV environment variable 93, 93
- USEOSTIME configuration parameter 93
- User environment variable 148
- USER function, in expression 311, 364
- User informix 9, 20, 134
- User name 80
- User privileges
  - syscolauth data 22
  - sysfragauth data 36
  - syslangauth data 46
  - sysprocauth data 50
  - systabauth data 64
  - sysusers data 71
  - sysxdttypeauth data 74
- User-defined aggregates 13
- User-defined casts 136
- User-defined casts (UDCs) 20
- User-defined data types
  - casting 136
  - casting into built-in type 133
  - opaque 132
  - sysxtddesc data 74
  - sysxtypes data 74, 75
- User-defined routines
  - casts (syscasts) 20
  - check constraints (syscheckudrdep) 21
  - error messages (syserrors) 34
  - for OPAQUE data types 108
  - functional index 172
  - language authorization (syslangauth) 46
  - privileges 50, 200

- protected 52
  - secondary access method 28
  - sysprocedures data 52
- USETABLENAME environment variable 215
- Using the GROUP BY and HAVING Clauses 320
- UTC time and time zone, returned by DBINFO function 313
- Utilities
  - chkenv 141, 144
  - DB-Access 9, 77, 145, 153, 160, 193
  - dbload 87, 89
  - dbschema 52, 52
  - env 144
  - export 143
  - gcc 187
  - getenv 141
  - ifx\_getenv 145
  - ifx\_putenv 145
  - imcadmin 186, 186
  - lp 165
  - lpr 165
  - MaxConnect 187
  - oninit 175
  - onload 87, 89
  - onpload 206
  - onsecurity 181
  - printenv 144
  - putenv 141
  - set 146
  - setenv 143
  - Setnet32 145
  - source 141
  - unset 143
  - unsetenv 143, 172
  - vi 159
- Utilities dbload 115
- Utility program
  - onload 393
  - onunload 393

## V

- VALUES clause
  - in INSERT statement 363
  - in MERGE statement 364
- NULL values 367
- restrictions 364
- selected columns 365
- valid values 364
- VARCHAR data type
  - ([]), brackets
    - MATCHES range delimiters 117
  - CHAR data type
    - collation 117
  - Code sets
    - collation order 117
    - East Asian 117
  - Collation
    - VARCHAR data type 117
  - coltype code 23
  - defined 117
  - Locales
    - collation order 117
  - MATCHES operator 117
  - Multibyte characters
    - VARCHAR data type 117
  - nonprintable characters 117
  - SQL\_LOGICAL\_CHAR configuration parameter 117
  - storing numeric values 117
  - VARCHAR data type
    - collation 117

- multibyte characters 117
- Zero (0)
  - C null as terminator 117
- VARCHAR data type, using LENGTH function on 311
- Variable-length opaque data types 23
- Variable-length packets 182
- Variable-length UDT 75
- Variables
  - defining and using in SPL routine 465
  - scope in SPL routine 466
  - with same name as a keyword 471
- VARIANCE function, as aggregate function 296
- VARIANT routine 52
- variant SPL function 460
- Version number, returned by DBINFO function 313
- Version of a table 65
- vi text editor 159
- View
  - columns view 78
  - definition of 227
  - deleting in a 532
  - Information Schema 76
  - inserting into a 532
  - INSTEAD OF trigger on a 533
  - server\_info view 80
  - sql\_languages view 79
  - sysdepend data 30
  - sysindexes view 43
  - sys synonyms data 62
  - sys syntable data 63
  - systabauth data 64
  - systables data 65
  - sysviews data 72
  - tables view 78
  - updating in a 532
- Violation detection 384
- Violations
  - sysobjstate data 47
  - sysviolations data 72
- Violations table 388
  - assigning a name 388
  - description of 388
  - example of privileges 389
  - examples 385
  - examples of starting 388
  - starting 388
- Virtual machine 148, 199
- Virtual processors 210

## W

- Warning message 34, 153
- Warnings, with SPL routine at compile time 515
- WEEKDAY function
  - as time function 297, 299
  - using 299
- WHERE 115
- WHERE clause
  - Boolean expression in 253
  - comparison condition 247
  - date-oriented functions in 299
  - description of 247
  - equal sign relational operator 248
  - host variables in 407
  - in DELETE 359
  - in UPDATE statement 372
  - less-than relational operator 249
  - not-equal relational operator 249

- relational operators 247
- selecting a range of characters 258
- subqueries in 339
- wildcard comparisons 254
- with NOT keyword 250
- with OR keyword 251
- WHERE CURRENT OF clause
  - in DELETE statement 425
  - in UPDATE statement 431
- WHERE keyword 9, 20
- null data tests 252
- range of values 249
- Whitespace in identifiers 172
- Wildcard character
  - asterisk 238
  - protecting 258
- Wildcard comparison in WHERE clause 254
- Wildcard, using single character 255
- Window borders 195
- Windows environments
  - manipulating environment variables 145
  - setting environment variables 145
- WITH clause, in SPL function 460
- WITH HOLD keywords, declaring a hold cursor 451
- WITH LISTING IN clause, use in SPL routine 461

## X

- X setting
  - sysams.am\_sptype 14
  - systabauth.tabauth 64
- X/Open
  - compliance 80
  - server\_info view 80
- X/Open CAE standards 76
- XA data source types 73
- XA data sources 73
- XML (Extensible Markup Language) 91
- XPG4 standard 78, 78

## Y

- Y setting
  - DBDATE 157
  - DBTIME 167
  - sysroleauth.is\_grantable 56
- Year 2000 154
- YEAR function
  - as time function 297
  - using 300
- YEAR keyword
  - DATETIME qualifier 93
  - EXTEND function 126
  - INTERVAL qualifier 101
- Year values, two and four digit 93, 154, 157, 167
- yes setting
  - NODEFDAC 200
- YES setting
  - columns.is\_nullable 78
  - sql\_languages.integrity 79

## Z

- Zero
  - extent size encoding 43
- Zero (0)
  - DBDATE separator 157
  - DECIMAL scale 96
  - hexadecimal digit 159
  - IFX\_DIRECTIVES setting 176, 177
  - IFX\_LARGE\_PAGES setting 178
  - IFX\_LONGID setting 179

- IFX\_NETBUF\_PVTPPOOL\_SIZE setting 180
- INFORMIXOPCACHE setting 192
- integer scale 78, 96
- OPTCOMPIND setting 202
- OPTMSG setting 202
- padding of 1-digit years 154
- padding with DBFLTMASK 160
- padding with DBTIME 167
- PDQPRIORITY setting 204
- PSORT\_NPROCS setting 210
- STMT\_CACHE setting 212
- sysams values 14, 14, 14, 14, 14
- sysfragments.hybdpos 39
- sysindices.nrows 43
- systables.type\_xid 65
- sysxdtypes values 75