HCL OneDB 2.0.1

OneDB Database Extensions User's Guide

Contents

Chapter 1. Database Extensions User's Guide

The *HCL OneDB™ Database Extensions User's Guide* explains how to use the database extensions that come with HCL OneDB™: Large Object Locator, MQ messaging, binary data types, basic text search, node data type, HCL OneDB™ web feature service for Geospatial Data, and SQL packages.

The topics contain the following information:

- The Large Object Locator extension manages large object data that is stored outside the database so that you can create a single consistent interface to large objects.
- The MQ messaging extension provides an interface to the HCL OneDB™ MQ (WMQ) messaging products. WMQ provides an infrastructure for distributed, asynchronous communication of data in a distributed, heterogeneous environment.
- The binary data types extension provides data types so that you can store binary-encoded strings, which can be indexed for quick retrieval.
- The basic text search extension provides a customizable index so that you can search words and phrases that are stored in a column of a table.
- The node data type extension gives you the ability to represent hierarchical data within the relational database.
- The SQL packages extension provides SPL routines that you can use in an application that is compatible with other database servers. For example, the packages include large object handling, alert and message management, and random number generation.

This publication is for application developers and database administrators who want to use the built-in extensions that are provided in HCL OneDB™ for storing, querying, and manipulating data.

Large object management

The Large Object Locator extension enables you to create a single consistent interface to large objects. It extends the concept of large objects to include data stored outside the database.

stores large object data (data that exceeds a length of 255 bytes or contains non-ASCII characters) in columns in the database. You can access this data using standard SQL statements. The server also provides functions for copying data between large object columns and files. See *HCL OneDB™ Guide to SQL: Syntax* and *HCL OneDB™ Guide to SQL: Tutorial* for more information.

With Large Object Locator you create a reference to a large object and store the reference as a row in the database. The object itself can reside outside the database: for example, on a file system (or it could be a BLOB or CLOB type column in the database). The reference identifies the type, or access protocol, of the object and points to its storage location. For example, you could identify an object as a file and provide a path name to it or identify it as a binary or character smart large object stored in the database. Smart large objects are a category of large objects that include CLOB and BLOB data types, which store text and images. Smart large objects are stored and retrieved in pieces, and have database properties such as crash recovery and transaction rollback.

You access a large object by passing its reference to a Large Object Locator function. For example, to open a large object for reading or writing, you pass the object's reference to the lld_open() function. This function uses the reference to find the location of the object and to identify its type. Based on the type, it calls the appropriate underlying function to open the object. For example, if the object is stored on a UNIX™ file system, lld_open() calls a UNIX™ function to open the object.

Important: In theory, you could use Large Object Locator to reference any type of large object in any storage location. In practice, access protocols must be built into Large Object Locator for each type of supported object. Because support for new types can be added at any time, be sure to read the release notes accompanying this publication—not the publication itself—to see the types of large objects Large Object Locator currently supports.

About Large Object Locator

Large Object Locator is implemented through two data types and a set of functions

The Large Object Locator data types are lld_locator and lld_lob.

You use the lld_locator type to identify the access protocol for a large object and to point to its location. This type is a row type, stored as a row in the database. You can insert, select, delete, and update instances of lld_locator rows in the database using standard SQL INSERT, SELECT, DELETE, and UPDATE statements.

You can also pass an lld_locator row to various Large Object Locator functions. For example, to create, delete, or copy a large object, and to open a large object for reading or writing, you pass an lld_locator row to the appropriate Large Object Locator function. See The lld_locator data type on page 6 for a detailed description of this data type.

The lld_lob type enables Large Object Locator to reference smart large objects, which are stored as BLOB or CLOB data in the database. The lld_lob type is identical to the BLOB and CLOB types except that, in addition to pointing to the data, it tracks whether the underlying smart large object contains binary or character data.

See The lld_lob data type on page 8 for a complete description of this data type.

Large Object Locator provides a set of functions similar to UNIX™ I/O functions for manipulating large objects. You use the same functions regardless of how or where the underlying large object is stored.

The Large Object Locator functions can be divided into four main categories:

Basic functions

Creating, opening, closing, deleting, and reading from and writing to large objects.

Client functions

Creating, opening, and deleting client files and for copying large objects to and from client files. After you open a client file, you can use the basic functions to read from and write to the file.

Utility functions

Raising errors and converting errors to their SQL state equivalents.

Smart large object functions

Copying smart large objects to files and to other smart large objects

There are three interfaces to the Large Object Locator functions:

- An API library
- An ESQL/C library
- An SQL interface

All Large Object Locator functions are implemented as API library functions. You can call Large Object Locator functions from user-defined routines within an application you build.

All Large Object Locator functions, except lld_error_raise(), are implemented as ESQL/C functions. You can use the Large Object Locator functions to build ESQL/C applications.

A limited set of the Large Object Locator functions are implemented as user-defined routines that you can execute within SQL statements. See SQL interface on page 10 for a list of the Large Object Locator functions that you can execute directly in SQL statements.

Large Object Locator functions on page 9, describes all the Large Object Locator functions and the three interfaces in detail.

Large object requirements

To implement the Large Object Locator, the Scheduler must be running and the database must conform to requirements. Certain limitations are inherent in using large objects with a database, because the objects themselves, except for smart large objects, are not stored in the database and are not subject to direct control by the server. Two specific areas of concern are transaction rollback and concurrency control.

Database server requirements

The HCL OneDB™ database server has the following requirements:

- Non-logged databases are not supported.
- ANSI databases are not supported.
- The Scheduler must be running.

If you attempt to create a Large Object Locator data type or run a Large Object Locator function in an unlogged or ANSI database, a message that `DataBlade registration failed` is printed in the online message log. If the Scheduler is not running the first time that you create a Large Object Locator data type or run a Large Object Locator function, a message that the data type is not found or the routine cannot be resolved is returned.

Transaction rollback

Because large objects, other than smart large objects, are stored outside the database, any changes to them take place outside the server's control and cannot be rolled back if a transaction is aborted. For example, when you execute lld_create(),

it calls an operating system routine to create the large object itself. If you roll back the transaction containing the call to lld_create(), the server has no way of deleting the object that you have just created.

Therefore, you are responsible for cleaning up any resources you have allocated if an error occurs. For example, if you create a large object and the transaction in which you create it is aborted, you should delete the object you have created. Likewise, if you have opened a large object and the transaction is aborted (or is committed), you should close the large object.

Concurrency control

Large Object Locator provides no direct way of controlling concurrent access to large objects. If you open a large object for writing, it is possible to have two separate processes or users simultaneously alter the large object. You must provide a means, such as locking a row, to guarantee that multiple users cannot access a large object simultaneously for writing.

Large Object Locator data types

This chapter describes the Large Object Locator data types, lld_locator and lld_lob.

The lld_locator data type

The lld_locator data type identifies a large object. It specifies the kind of large object and provides a pointer to its location. lld_locator is a row type and is defined as follows:

```
create row type informix.lld_locator
   {
   lo_protocol                  char(18
   lo_pointer           informix.lld_lob0
   lo_location           informix.lvarchar
   }
```

lo_protocol

Identifies the kind of large object.

lo_pointer

A pointer to a smart large object, or is `NULL` if the large object is any kind of large object other than a smart large object.

lo_location

A pointer to the large object, if it is not a smart large object. Set to `NULL` if it is a smart large object.

In the *lo_protocol* field, specify the kind of large object to create. The kind of large object you specify determines the values of the other two fields:

- If you specify a smart large object:
  - use the *lo_pointer* field to point to it.
  - specify `NULL` for the *lo_location* field.
- If you specify any other kind of large object:
  - specify `NULL` for the *lo_pointer* field.
  - use the *lo_location* field to point to it.

The *lo_pointer* field uses the lld_lob data type, which is defined by Large Object Locator. This data type allows you to point to a smart large object and specify whether it is of type BLOB or type CLOB. For more information, see The lld_lob data type on page 8.

The *lo_location* field uses an lvarchar data type, which is a varying-length character type.

The following table lists the current protocols and summarizes the values for the other fields based on the protocol that you specify. Be sure to check the release notes shipped with this publication to see if Large Object Locator supports additional protocols not listed here.

Tip: Although the lld_locator type is not currently extensible, it might become so later. To avoid future name space collisions, the protocols established by Large Object Locator all have an IFX prefix.

Table 1. Fields of lld_locator data type

lo_protocol	lo_pointer	lo_location	Description
IFX_BLOB	Pointer to a smart large object	NULL	Smart large object
IFX_CLOB	Pointer to a smart large object	NULL	Smart large object
IFX_FILE	NULL	pathname	File accessible on server

Important: The lo_protocol field is not case-sensitive. It is shown in uppercase letters for display purposes only.

The lld_locator type is an instance of a row type. You can insert a row into the database using an SQL INSERT statement, or you can obtain a row by calling the DataBlade® API mi_row_create() function. See the *HCL OneDB™ ESQL/C Programmer's Manual* for information about row types. See the *HCL OneDB™ DataBlade® API Programmer's Guide* for information about the mi_row_create() function.

To reference an existing large object, you can insert an lld_locator row directly into a table in the database.

To create a large object, and a reference to it, you can call the lld_create() function and pass an lld_locator row.

You can pass an lld_locator type to these Large Object Locator functions, described in Large Object Locator functions on page 9:

- The lld_copy() function on page 12
- The lld_create() function on page 14
- The lld_delete() function on page 16
- The lld_open() function on page 17
- The lld_from_client() function on page 26
- The lld_to_client() function on page 30

The lld_lob data type

The lld_lob data type is a user-defined type. You can use it to specify the location of a smart large object and to specify whether the object contains binary or character data.

The lld_lob data type is defined for use with the API as follows:

```
typedef struct
   {
   MI_LO_HANDLE                lo;
   mi_integer                  type;
   } lld_lob_t;
```

It is defined for ESQL/C as follows:

```
typedef struct
   {
   ifx_lo_t                    lo;
   int                         type;
   } lld_lob_t;
```

lo

A pointer to the location of the smart large object.

type

The type of the object. For an object containing binary data, set *type* to LLD_BLOB; for an object containing character data, set *type* to LLD_CLOB.

The lld_lob type is equivalent to the CLOB or BLOB type in that it points to the location of a smart large object. In addition, it specifies whether the object contains binary or character data. You can pass the lld_lob type as the *lo_pointer* field of an lld_locator row. You should set the **lld_lob_t.***type* field to LLD_BLOB for binary data and to LLD_CLOB for character data.

See The lld_lob type on page 34 for example code that uses the lld_lob type.

LOB Locator provides explicit casts from:

- a CLOB type to an lld_lob type.
- a BLOB type to an lld_lob type.
- an lld_lob type to the appropriate BLOB or CLOB type.

Tip: If you attempt to cast an lld_lob type containing binary data into a CLOB type or an lld_lob type containing character data into a BLOB type, Large Object Locator returns an error message.

You can pass an lld_lob type to these functions, described in Large Object Locator functions on page 9:

- The LOCopy function on page 32
- The LOToFile function on page 33
- The LLD_LobType function on page 34

Note that LOCopy and LOToFile functions are overloaded versions of built-in server functions. The only difference is that you pass an lld_lob to the Large Object Locator versions of these functions and a BLOB or CLOB type to the built-in versions.

Large Object Locator functions

This chapter briefly describes the three interfaces to Large Object Locator and describes in detail all the Large Object Locator functions.

Interfaces

Large Object Locator functions are available through three interfaces:

- An API library
- An ESQL/C library
- An SQL interface

If the syntax for a function depends on the interface, each syntax appears under a separate subheading. Because there are few differences between parameters and usage in the different interfaces, there is a single parameter description and one *"Usage," "Return,"* and *"Related topics"* section for each function. Where there are differences between the interfaces, these differences are described.

The naming convention for the SQL interface is different from that for the ESQL/C and API interfaces. For example, the SQL client copy function is called LLD_ToClient(), whereas the API and ESQL/C client copy functions are called lld_to_client(). This publication uses the API and ESQL/C naming convention unless referring specifically to an SQL function.

API library

All Large Object Locator functions except the smart large object functions are implemented as API functions defined in header and library files (`lldsapi.h` and `lldsapi.a`).

You can call the Large Object Locator API functions from your own user-defined routines. You execute Large Object Locator API functions just as you do functions provided by the HCL® OneDB® DataBlade® API. See the *HCL OneDB™ DataBlade® API Programmer's Guide* for more information.

See The API interface on page 40 for an example of a user-defined routine that calls Large Object Locator API functions to copy part of a large object to another large object.

ESQL/C library

All Large Object Locator functions except lld_error_raise() and the smart large object functions are implemented as ESQL/C functions, defined in header and library files (`lldesql.h` and `lldesql.so`).

Wherever possible, the ESQL/C versions of the Large Object Locator functions avoid server interaction by directly accessing the underlying large object.

See the *HCL OneDB™ ESQL/C Programmer's Manual* for more information about using the ESQL/C interface to execute Large Object Locator functions.

SQL interface

The following Large Object Locator functions are implemented as user-defined routines that you can execute within SQL statements:

- LLD_LobType()
- LLD_Create()
- LLD_Delete()
- LLD_Copy()
- LLD_FromClient()
- LLD_ToClient()
- LOCopy()
- LOToFile()

See the following three-volume set for further information about the HCL OneDB™ SQL interface:

- *HCL OneDB™ Guide to SQL: Reference*
- *HCL OneDB™ Guide to SQL: Syntax*
- *HCL OneDB™ Guide to SQL: Tutorial*

Working with large objects

This section describes functions that allow you to:

- create large objects.
- open, close, and delete large objects.
- return and change the current position within a large object.
- read from and write to large objects.
- copy a large object.

Generally, you use the functions described in this section in the following order.

1. You use **lld_create()** to create a large object. It returns a pointer to an lld_locator row that points to the large object.

   If the large object already exists, you can insert an lld_locator row into a table in the database to point to the object without calling **lld_create()**.

2. You can pass the lld_locator type to the **lld_open()** function to open the large object you created. This function returns an **LLD_IO** structure that you can pass to various Large Object Locator functions to manipulate data in the open object (see Step 3 on page 11).

   You can also pass the lld_locator type to the **lld_copy(), lld_from_client()**, or **lld_to_client()** functions to copy the large object.

3. After you open a large object, you can pass the **LLD_IO** structure to:

   lld_tell()

   Returns the current position within the large object.

   lld_seek()

   Changes the current position within the object.

   lld_read()

   Reads from large object.

   lld_write()

   Writes to the large object.

   lld_close()

   Closes an object. You should close a large object if the transaction in which you open it is aborted or committed.

Tip: To delete a large object, you can pass the lld_locator row to **lld_delete()** any time after you create it. For example, if the transaction in which you created the object is aborted and the object is not a smart large object, you should delete the object because the server's rollback on the transaction cannot delete an object outside the database.

The functions within this section are presented in alphabetical order, not in the order in which you might use them.

The lld_close() function

This function closes the specified large object.

Syntax

API

```
mi_integer lld_close (conn, io, error)
   MI_CONNECTION*              conn;
   LLD_IO*                     io;
   mi_integer*                 error;
```

ESQL/C

```
int lld_close (LLD_IO* io, int* error);
```

conn

The connection descriptor established by a previous call to the mi_open() or mi_server_connect() functions. This parameter is for the API interface only. In the ESQL/C version of this function, you must already be connected to a server.

io

A pointer to an **LLD_IO** structure created with a previous call to the lld_open() function.

error

An output parameter in which the function returns an error code.

Usage

The lld_close() function closes the open large object and frees the memory allocated for the **LLD_IO** structure, which you cannot use again after this call.

Return codes

For an API function, returns `MI_OK` if the function succeeds and `MI_ERROR` if it fails.

For an ESQL/C function, returns `0` if the function succeeds and `-1` if it fails.

Context

The lld_open() function on page 17

The lld_copy() function

This function copies the specified large object.

Syntax

API

```
MI_ROW* lld_copy(conn, src, dest, error);
   MI_CONNECTION*              conn,
   MI_ROW*                     src,
   MI_ROW*                     dest,
   mi_integer*                 error
```

ESQL/C

```
ifx_collection_t* lld_copy (src, dest, error);
   EXEC SQL BEGIN DECLARE SECTION;
      PARAMETER ROW src;
      PARAMETER ROW dest;
   EXEC SQL END DECLARE SECTION;
   int* error;
```

SQL

```
CREATE FUNCTION LLD_Copy (src LLD_Locator, dest LLD_Locator)
   RETURNS LLD_Locator;
```

conn

The connection descriptor established by a previous call to the mi_open() or mi_server_connect() function. This parameter is for the API interface only. In the ESQL/C and SQL versions of this function, you must already be connected to a server.

src

A pointer to the lld_locator row, identifying the source object.

dest

A pointer to an lld_locator row, identifying the destination object. If the destination object itself does not exist, it is created.

error

An output parameter in which the function returns an error code. The SQL version of this function does not have an *error* parameter.

Usage

This function copies an existing large object.

If the destination object exists, pass a pointer to its lld_locator row as the *dest* parameter.

If the destination object does not exist, pass an lld_locator row with the following values as the *dest* parameter to lld_copy():

In the *lo_protocol* field, specify the type of large object to create.

If you are copying to any type of large object other than a smart large object:

- specify `NULL` for the *lo_pointer* field.
- point to the location of the new object in the *lo_location* field.

The lld_copy() function creates the type of large object that you specify, copies the source object to it, and returns the row you passed, unaltered.

If you are copying to a smart large object, specify `NULL` for the *lo_pointer* and *lo_location* fields of the lld_locator row that you pass as the *dest* parameter. The lld_copy() function returns an lld_locator row with a pointer to the new smart large object in the *lo_pointer* field.

The server deletes a new smart large object at the end of a transaction if there are no disk references to it and if it is closed. Therefore, after copying to a newly created smart large object, either open it or insert it into a table.

If lld_copy() creates a new smart large object, it uses system defaults for required storage parameters such as *sbspace*. If you want to override these parameters, you can use the server large object interface to create the smart large object and specify the parameters you want in an **MI_LO_SPEC** structure. You can then call lld_copy() and set the *lo_pointer* field of the lld_locator row to point to the new smart large object.

Likewise, if protocols are added to Large Object Locator for new types of large objects, these objects might require creation attributes or parameters for which Large Object Locator supplies predefined default values. As with smart large objects, you can create the object with lld_copy() and accept the default values, or you can use the creation routines specific to the new protocol and supply your own attributes and parameters. After you create the object, you can call lld_copy() and pass it an lld_locator row that points to the new object.

Return codes

On success, this function returns a pointer to an lld_locator row, specifying the location of the copy of the large object. If the destination object already exists, lld_copy() returns a pointer to the unaltered lld_locator row you passed in the *dest* parameter. If the destination object does not already exist, lld_copy() returns a pointer to an lld_locator row, pointing to the new object it creates.

On failure, this function returns `NULL`.

Context

The lld_from_client() function on page 26

The lld_to_client() function on page 30

The lld_create() function

This function creates a new large object with the protocol and location you specify.

Syntax

API

```
MI_ROW* lld_create(conn, lob, error)
   MI_CONNECTION*              conn
   MI_ROW*                     lob;
   mi_integer*                 error;
```

ESQL/C

```
ifx_collection_t* lld_create (lob, error);
   EXEC SQL BEGIN DECLARE SECTION;
      PARAMETER ROW lob;
   EXEC SQL END DECLARE SECTION;
   int* error;
```

SQL

```
CREATE FUNCTION LLD_Create (lob LLD_Locator)
   RETURNS LLD_Locator;
```

conn

The connection descriptor established by a previous call to the mi_open() or mi_server_connect() functions. This parameter is for the API interface only. In the ESQL/C and SQL versions of this function, you must already be connected to a server.

lob

A pointer to an lld_locator row, identifying the object to create.

error

An output parameter in which the function returns an error code. The SQL version of this function does not have an *error* parameter.

Usage

You pass an lld_locator row, with the following values, as the *lob* parameter to lld_create():

In the *lo_protocol* field, specify the type of large object to create.

For any type of large object other than a smart large object:

- specify `NULL` for the *lo_pointer* field.
- point to the location of the new object in the *lo_location* field.

The lld_create() function returns the row you passed, unaltered.

If you are creating a smart large object, specify `NULL` for the *lo_pointer* and *lo_location* fields of the lld_locator row. The lld_create() function returns an lld_locator row with a pointer to the new smart large object in the *lo_pointer* field.

The server deletes a new smart large object at the end of a transaction if there are no disk references to it and if it is closed. Therefore, after creating a smart large object, either open it or insert it into a table.

Large Object Locator does not directly support transaction rollback, except for smart large objects. Therefore, if the transaction in which you call lld_create() is aborted, you should call lld_delete() to delete the object and reclaim any allocated resources.

See Large object requirements on page 5 for more information.

When you create a smart large object, lld_create() uses system defaults for required storage parameters such as *sbspace*. If you want to override these parameters, you can use the server large object interface to create the smart large object and specify the parameters you want in an **MI_LO_SPEC** structure. You can then call lld_create() and set the *lo_pointer* field of the lld_locator row to point to the new smart large object.

Likewise, if protocols are added to Large Object Locator for new types of large objects, these objects might require creation attributes or parameters for which Large Object Locator supplies predefined default values. As with smart large objects, you can create the object with lld_create() and accept the default values, or you can use the creation routines specific to the new protocol and supply your own attributes and parameters. After you create the object, you can call lld_create() and pass it an lld_locator row that points to the new object.

Return codes

On success, this function returns a pointer to an lld_locator row specifying the location of the new large object. For a smart large object, lld_create() returns a pointer to the location of the new object in the *lo_pointer* field of the lld_locator row. For all other objects, it returns a pointer to the unaltered lld_locator row you passed in the *lob* parameter.

The lld_open function can use the lld_locator row that lld_create() returns.

On failure, this function returns `NULL`.

Context

The lld_delete() function on page 16

The lld_open() function on page 17

The lld_delete() function

This function deletes the specified large object.

Syntax

API

```
mi_integer lld_delete(conn, lob, error)
   MI_CONNECTION*              conn;
   LLD_Locator                 lob;
   mi_integer*                 error;
```

ESQL/C

```
int lld_delete (lob, error);
EXEC SQL BEGIN DECLARE SECTION;
   PARAMETER ROW lob;
EXEC SQL END DECLARE SECTION;
   int* error;
```

SQL

```
CREATE FUNCTION LLD_Delete (lob LLD_Locator)
   RETURNS BOOLEAN;
```

conn

The connection descriptor established by a previous call to the mi_open() or mi_server_connect() functions. This parameter is for the API interface only. In the ESQL/C and SQL versions of this function, you must already be connected to a server.

lob

A pointer to an lld_locator row, identifying the object to delete.

error

An output parameter in which the function returns an error code. The SQL version of this function does not have an *error* parameter.

Usage

For large objects other than smart large objects, this function deletes the large object itself, not just the lld_locator row referencing it. For smart large objects, this function does nothing.

To delete a smart large object, delete all references to it, including the lld_locator row referencing it.

Return codes

For an API function, returns `MI_OK` if the function succeeds and `MI_ERROR` if it fails.

For an ESQL/C function, returns `0` if the function succeeds and `-1` if the function fails.

The lld_open() function

This function opens the specified large object.

Syntax

API

```
LLD_IO* lld_open(conn, lob, flags, error)
   MI_CONNECTION*              conn;
   MI_ROW*                     lob;
   mi_integer                  flags,
   mi_integer*                 error);
```

ESQL/C

```
LLD_IO* lld_open(lob, flags, error);
   EXEC SQL BEGIN DECLARE SECTION;
      PARAMETER ROW lob;
   EXEC SQL END DECLARE SECTION;
      int flags;int* error;
```

conn

The connection descriptor established by a previous call to the mi_open() or mi_server_connect() functions. This parameter is for the API interface only. In the ESQL/C and SQL versions of this function, you must already be connected to a server.

lob

A pointer to an lld_locator row, identifying the object to delete.

flags

A set of flags that you can set to specify attributes of the large object after it is opened. The flags are as follows:

LLD_RDONLY

Opens the large object for reading only. You cannot use the lld_write function to write to the specified large object when this flag is set.

LLD_WRONLY

Opens the large object for writing only. You cannot use the lld_read() function to read from the specified large object when this flag is set.

LLD_RDWR

Opens the large object for both reading and writing.

LLD_TRUNC

Clears the contents of the large object after opening.

LLD_APPEND

Seeks to the end of the large object for writing. When the object is opened, the file pointer is positioned at the beginning of the object. If you have opened the object for reading or reading and writing, you can seek anywhere in the file and read. However, any time you call lld_write() to write to the object, the pointer moves to the end of the object to guarantee that you do not overwrite any data.

LLD_SEQ

Opens the large object for sequential access only. You cannot use the lld_seek() function with the specified large object when this flag is set.

error

An output parameter in which the function returns an error code.

Usage

In the *lob* parameter, you pass an lld_locator row to identify the large object to open. In the *lo_protocol* field of this row, you specify the type of the large object to open. The lld_open() function calls an appropriate open routine based on the type you specify. For example, for a file, lld_open() uses an operating system file function to open the file, whereas, for a smart large object, it calls the server's mi_lo_open() routine.

Large Object Locator does not directly support two fundamental database features, transaction rollback and concurrency control. Therefore, if the transaction in which you call lld_open() is aborted, you should call lld_close() to close the object and reclaim any allocated resources.

Your application should also provide some means, such as locking a row, to guarantee that multiple users cannot write to a large object simultaneously.

See Large object requirements on page 5 for more information about transaction rollback and concurrency control.

Return codes

On success, this function returns a pointer to an **LLD_IO** structure it allocates. The **LLD_IO** structure is private, and you should not directly access it or modify its contents. Instead, you can pass the **LLD_IO** structure's pointer to Large Object Locator routines such as lld_write(), lld_read(), and so on, that access open large objects.

A large object remains open until you explicitly close it with the lld_close() function. Therefore, if you encounter error conditions after opening a large object, you are responsible for reclaiming resources by closing it.

On failure, this function returns `NULL`.

Context

The lld_close() function on page 11

The lld_create() function on page 14

The lld_read() function on page 19

The lld_seek() function on page 20

The lld_tell() function on page 21

The lld_write() function on page 22

The lld_read() function

This function reads from a large object, starting at the current position.

Syntax

API

```
mi_integer lld_read (io, buffer, bytes, error)

LLD_IO*                         io,
void*                           buffer,
mi_integer                      bytes,
mi_integer*                     error);
```

ESQL/C

```
int lld_read (LLD_IO* io,
            void* buffer, int bytes,
            int* error);
```

io

A pointer to an **LLD_IO** structure created with a previous call to the lld_open() function.

buffer

A pointer to a buffer into which to read the data. The buffer must be at least as large as the number of bytes specified in the *bytes* parameter.

bytes

The number of bytes to read.

error

An output parameter in which the function returns an error code.

Usage

Before calling this function, you must open the large object with a call to lld_open() and set the LLD_RDONLY or LLD_RDWR flag. The lld_read() function begins reading from the current position. By default, when you open a large object, the current position is the beginning of the object. You can call lld_seek() to change the current position.

Return codes

On success, the lld_read() function returns the number of bytes that it has read from the large object.

On failure, for an API function, it returns `MI_ERROR`; for an ESQL/C function, it returns `-1`.

Context

The lld_open() function on page 17

The lld_seek() function on page 20

The lld_tell() function on page 21

The lld_seek() function

This function sets the position for the next read or write operation to or from a large object that is open for reading or writing.

Syntax

API

```
mi_integer lld_seek(conn, io, offset, whence, new_offset, error)
   MI_CONNECTION*               conn
   LLD_IO*                      io;
   mi_int8*                     offset;
   mi_integer                   whence;
   mi_int8*                     new_offset;
   mi_integer*                  error;
```

ESQL/C

```
int lld_seek(io,offset, whence, new_offset, error)
   LLD_IO* io;
EXEC SQL BEGIN DECLARE SECTION;
   PARAMETER int8* offset;
EXEC SQL END DECLARE SECTION;
EXEC SQL BEGIN DECLARE SECTION;
   PARAMETER int8* new_offset;
EXEC SQL END DECLARE SECTION;
   int whence;
   int* error;
```

conn

The connection descriptor established by a previous call to the mi_open() or mi_server_connect() functions. This parameter is for the API interface only. In the ESQL/C and SQL versions of this function, you must already be connected to a server.

io

A pointer to an **LLD_IO** structure created with a previous call to the lld_open() function.

offset

A pointer to the offset. It describes where to seek in the object. Its value depends on the value of the *whence* parameter.

- If *whence* is LLD_SEEK_SET, the offset is measured relative to the beginning of the object.
- If *whence* is LLD_SEEK_CUR, the offset is relative to the current position in the object.
- If *whence* is LLD_SEEK_END, the offset is relative to the end of the file.

whence

Determines how the offset is interpreted.

new_offset

A pointer to an **int8** that you allocate. The function returns the new offset in this **int8**.

error

An output parameter in which the function returns an error code.

Usage

Before calling this function, you must open the large object with a call to lld_open().

Although this function takes an 8-byte offset, this offset is converted to the appropriate size for the underlying large object storage system. For example, if the large object is stored in a 32-bit file system, the 8-byte offset is converted to a 4-byte offset, and any attempt to seek past 4 GB generates an error.

Return codes

For an API function, returns `MI_OK` if the function succeeds and `MI_ERROR` if it fails.

For an ESQL/C function, returns `0` if the function succeeds and `-1` if the function fails.

Context

The lld_open() function on page 17

The lld_read() function on page 19

The lld_tell() function on page 21

The lld_write() function on page 22

The lld_tell() function

This function returns the offset for the next read or write operation on an open large object.

Syntax

API

```
mi_integer lld_tell(conn, io, offset, error)
   MI_CONNECTION*               conn;
   LLD_IO*                      io,
   mi_int8*                     offset;
   mi_integer*                  error;
```

ESQL/C

```
int lld_tell (io, offset, error);
   LLD_IO* io;
EXEC SQL BEGIN DECLARE SECTION;
   PARAMETER int8* offset;
EXEC SQL END DECLARE SECTION;
   int* error;
```

conn

The connection descriptor established by a previous call to the mi_open() or mi_server_connect() functions. This parameter is for the API interface only. In the ESQL/C and SQL versions of this function, you must already be connected to a server.

io

A pointer to an **LLD_IO** structure created with a previous call to the lld_open() function.

offset

A pointer to an **int8** that you allocate. The function returns the offset in this **int8**.

error

An output parameter in which the function returns an error code.

Usage

Before calling this function, you must open the large object with a call to lld_open().

Return codes

For an API function, returns `MI_OK` if the function succeeds and `MI_ERROR` if it fails.

For an ESQL/C function, returns `0` if the function succeeds and `-1` if the function fails.

Context

The lld_open() function on page 17

The lld_read() function on page 19

The lld_seek() function on page 20

The lld_write() function on page 22

The lld_write() function

This function writes data to an open large object, starting at the current position.

Syntax

API

```
mi_integer lld_write (conn, io, buffer, bytes, error)
   MI_CONNECTION*               conn;
   LLD_IO*                      io;
   void*                        buffer;
   mi_integer                   bytes;
   mi_integer*                  error;
```

ESQL/C

```
int lld_write (LLD_IO* io, void* buffer,
            int bytes, int* error);
```

conn

The connection descriptor established by a previous call to the mi_open() or mi_server_connect() functions. This parameter is for the API interface only. In the ESQL/C and SQL versions of this function, you must already be connected to a server.

io

A pointer to an **LLD_IO** structure created with a previous call to the lld_open() function.

buffer

A pointer to a buffer from which to write the data. The buffer must be at least as large as the number of bytes specified in the *bytes* parameter.

bytes

The number of bytes to write.

error

An output parameter in which the function returns an error code.

Usage

Before calling this function, you must open the large object with a call to lld_open() and set the LLD_WRONLY or LLD_RDWR flag. The lld_write() function begins writing from the current position. By default, when you open a large object, the current position is the beginning of the object. You can call lld_seek() to change the current position.

If you want to append data to the object, specify the LLD_APPEND flag when you open the object to set the current position to the end of the object. If you have done so and have opened the object for reading and writing, you can still use lld_seek to move around in the object and read from different places. However, as soon as you begin to write, the current position is moved to the end of the object to guarantee that you do not overwrite any existing data.

Return codes

On success, the lld_write() function returns the number of bytes that it has written.

On failure, for an API function it returns `MI_ERROR`; for an ESQL/C function, it returns `-1`.

Context

The lld_open() function on page 17

The lld_seek() function on page 20

The lld_tell() function on page 21

Client file support

This section describes the Large Object Locator functions that provide client file support. These functions allow you to create, open, and delete client files and to copy large objects to and from client files.

The client functions make it easier to code user-defined routines that input or output data. These user-defined routines, in many cases, operate on large objects. They also input data from or output data to client files. Developers can create two versions of a user-defined routine: one for client files, which calls lld_open_client(), and one for large objects, which calls lld_open(). After the large object or client file is open, you can use any of the Large Object Locator functions that operate on open objects, such as lld_read(), lld_seek(), and so on. Thus, the remaining code of the user-defined function can be the same for both versions.

You should use the Large Object Locator client functions with care. You can only access client files if you are using the client machine on which the files are stored. If you change client machines, you can no longer access files stored on the original client machine. Thus, an application that stores client file names in the database might find at a later date that the files are inaccessible.

The lld_create_client() function

This function creates a new client file.

Syntax

API

```
mi_integer lld_create_client(conn, path, error);
   MI_CONNECTION*               conn
   mi_string*                   path;
   mi_integer*                  error;
```

ESQL/C

```
int lld_create_client (char* path, int* error);
```

conn

The connection descriptor established by a previous call to the mi_open() or mi_server_connect() functions. This parameter is for the API interface only. In the ESQL/C and SQL versions of this function, you must already be connected to a server.

path

A pointer to the path name of the client file.

error

An output parameter in which the function returns an error code.

Usage

This function creates a file on your client machine. Use the lld_open_client() function to open the file for reading or writing and pass it the same pathname as you passed to lld_create_client().

Large Object Locator does not directly support transaction rollback, except for smart large objects. Therefore, if the transaction in which you call lld_create_client() is aborted, you should call lld_delete_client() to delete the object and reclaim any allocated resources.

See Large object requirements on page 5 for more information.

Return codes

For an API function, returns `MI_OK` if the function succeeds and `MI_ERROR` if it fails.

For an ESQL/C function, returns `0` if the function succeeds and `-1` if the function fails.

Context

The lld_delete_client() function on page 25

The lld_delete_client() function

This function deletes the specified client file.

Syntax

API

```
mi_integer lld_delete_client(conn, path, error)
   MI_CONNECTION*               conn;
   mi_string*                   path;
   mi_integer*                  error;
```

ESQL/C

```
int lld_delete_client (char* path,int* error);
```

conn

The connection descriptor established by a previous call to the mi_open() or mi_server_connect() functions. This parameter is for the API interface only. In the ESQL/C and SQL versions of this function, you must already be connected to a server.

path

A pointer to the path name of the client file.

error

An output parameter in which the function returns an error code.

Usage

This function deletes the specified client file and reclaims any allocated resources.

Return codes

For an API function, returns `MI_OK` if the function succeeds and `MI_ERROR` if it fails.

For an ESQL/C function, returns `0` if the function succeeds and `-1` if the function fails.

Context

The lld_create_client() function on page 24

The lld_from_client() function

This function copies a client file to a large object.

Syntax

API

```
MI_ROW* lld_from_client(conn, src, dest, error);
   MI_CONNECTION*              conn,
   mi_string*                  src,
   MI_ROW*                     dest,
   mi_integer*                 error
```

ESQL/C

```
ifx_collection_t* lld_from_client (src, dest, error);
   char* src;
   EXEC SQL BEGIN DECLARE SECTION;
      PARAMETER ROW dest;
   EXEC SQL END DECLARE SECTION;
   int* error;
```

SQL

```
CREATE FUNCTION LLD_FromClient(src LVARCHAR,
                  dest LLD_Locator)
   RETURNS LLD_Locator;
```

conn

The connection descriptor established by a previous call to the mi_open() or mi_server_connect() functions. This parameter is for the API interface only. In the ESQL/C and SQL versions of this function, you must already be connected to a server.

src

A pointer to the source path name.

dest

A pointer to the destination lld_locator row. If the destination object itself does not exist, it is created.

error

An output parameter in which the function returns an error code. The SQL version of this function does not have an *error* parameter.

Usage

This function copies an existing large object.

If the destination object exists, pass a pointer to its lld_locator row as the *dest* parameter.

If the destination object does not exist, pass an lld_locator row with the following values as the *dest* parameter to lld_from_client().

In the *lo_protocol* field, specify the type of large object to create.

If you are copying to any type of large object other than a smart large object:

- specify `NULL` for the *lo_pointer* field.
- point to the location of the new object in the *lo_location* field.

The lld_from_client() function creates the type of large object that you specify, copies the source file to it, and returns the row you passed, unaltered.

If you are copying to a smart large object, specify `NULL` for the *lo_pointer* and *lo_location* fields of the lld_locator row that you pass as the *dest* parameter. The lld_from_client() function returns an lld_locator row with a pointer to the new smart large object in the *lo_pointer* field.

The server deletes a new smart large object at the end of a transaction if there are no disk references to it and if it is closed. Therefore, after you copy to a newly created smart large object, either open it or insert it into a table.

If lld_from_client() creates a new smart large object, it uses system defaults for required storage parameters such as *sbspace*. If you want to override these parameters, you can use the server large object interface to create the smart large object and specify the parameters you want in an **MI_LO_SPEC** structure. You can then call lld_from_client() and set the *lo_pointer* field of the lld_locator row to point to the new smart large object.

Likewise, if protocols are added to Large Object Locator for new types of large objects, these objects might require creation attributes or parameters for which Large Object Locator supplies predefined default values. As with smart large objects, you can create the object with lld_from_client() and accept the default values, or you can use the creation routines specific to the new protocol and supply your own attributes and parameters. After you create the object, you can call lld_from_client() and pass it an lld_locator row that points to the new object.

Return codes

On success, returns a pointer to an lld_locator row that specifies the location of the copy of the large object. If the destination object already exists, lld_from_client() returns a pointer to the unaltered lld_locator row that you created and

passed in the *dest* parameter. If the destination object does not already exist, lld_from_client() returns an lld_locator row that points to the new object it creates.

On failure, this function returns `NULL`.

Context

The lld_create_client() function on page 24

The lld_open_client() function on page 28

The lld_open_client() function

This function opens a client file.

Syntax

API

```
LLD_IO* lld_open_client(conn, path, flags, error);
   MI_CONNECTION*              conn
   mi_string*                  path;
   mi_integer                  flags;
   mi_integer*                 error;
```

ESQL/C

```
LLD_IO* lld_open_client(MI_CONNECTION* conn,mi_string* path,
mi_integer flags,mi_integer* error);
```

conn

The connection descriptor established by a previous call to the mi_open() or mi_server_connect() functions. This parameter is for the API interface only. In the ESQL/C and SQL versions of this function, you must already be connected to a server.

path

A pointer to the path name of the client file.

flags

A set of flags that you can set to specify attributes of the large object after it is opened. The flags are as follows:

LLD_RDONLY

Opens the client file for reading only. You cannot use the lld_write function to write to the specified client file when this flag is set.

LLD_WRONLY

Opens the client file for writing only. You cannot use the lld_read() function to read from the specified client file when this flag is set.

LLD_RDWR

Opens the client file for both reading and writing.

LLD_TRUNC

Clears the contents of the client file after opening.

LLD_APPEND

Seeks to the end of the large object for writing. When the object is opened, the file pointer is positioned at the beginning of the object. If you have opened the object for reading or reading and writing, you can seek anywhere in the file and read. However, any time you call lld_write() to write to the object, the pointer moves to the end of the object to guarantee that you do not overwrite any data.

LLD_SEQ

Opens the client file for sequential access only. You cannot use the lld_seek() function with the specified client file when this flag is set.

error

An output parameter in which the function returns an error code.

Usage

This function opens an existing client file. After the file is open, you can use any of the Large Object Locator functions, such as lld_read(), lld_write(), and so on, that operate on open large objects.

Large Object Locator does not directly support two fundamental database features, transaction rollback and concurrency control. Therefore, if the transaction in which you call lld_open_client() is aborted, you should call lld_close() to close the object and reclaim any allocated resources.

Your application should also provide some means, such as locking a row, to guarantee that multiple users cannot write to a large object simultaneously.

See Large object requirements on page 5 for more information about transaction rollback and concurrency control.

Return codes

On success, this function returns a pointer to an **LLD_IO** structure that it allocates. The **LLD_IO** structure is private, and you should not directly access it or modify its contents. Instead, you should pass its pointer to Large Object Locator routines such as lld_write(), lld_read(), and so on, that access open client files.

A client file remains open until you explicitly close it with the lld_close() function. Therefore, if you encounter error conditions after opening a client file, you are responsible for reclaiming resources by closing it.

On failure, this function returns `NULL`.

Context

The lld_close() function on page 11

The lld_read() function on page 19

The lld_seek() function on page 20

The lld_tell() function on page 21

The lld_write() function on page 22

The lld_create_client() function on page 24

The lld_to_client() function

This function copies a large object to a client file.

Syntax

API

```
MI_ROW* lld_to_client(conn, src, dest, error);
   MI_CONNECTION*              conn,
   MI_ROW*                     src,
   mi_string*                  dest,
   mi_integer*                 error
```

ESQL/C

```
ifx_collection_t* lld_to_client (src, dest, error);
   EXEC SQL BEGIN DECLARE SECTION;
      PARAMETER ROW src;
   EXEC SQL END DECLARE SECTION;
   char* dest;
   int* error;
```

SQL

```
LLD_ToClient (src LLD_Locator, dest LVARCHAR)
   RETURNS BOOLEAN;
```

conn

The connection descriptor established by a previous call to the mi_open() or mi_server_connect() functions. This parameter is for the API interface only. In the ESQL/C and SQL versions of this function, you must already be connected to a server.

src

A pointer to the lld_locator row that identifies the source large object.

dest

A pointer to the destination path name. If the destination file does not exist, it is created.

error

An error code. The SQL version of this function does not have an *error* parameter.

Usage

This function copies an existing large object to a client file. It creates the client file if it does not already exist.

Return codes

For an API function, returns `MI_OK` if the function succeeds and `MI_ERROR` if it fails.

For an ESQL/C function, returns `0` if the function succeeds and `-1` if the function fails.

Context

The lld_open_client() function on page 28

Error utility functions

The two functions described in this section allow you to raise error exceptions and convert error codes to their SQL state equivalent.

The lld_error_raise() function

This function generates an exception for the specified error.

Syntax

API

```
mi_integer lld_error_raise (error);
   mi_integer                  error
```

error

An error code that you specify.

Usage

This function calls the server mi_db_error_raise function to generate an exception for the specified Large Object Locator error.

Return codes

On success, this function does not return a value unless the exception is handled by a callback function. If the exception is handled by the callback and control returns to lld_error_raise(), it returns `MI_ERROR`.

On failure, it also returns `MI_ERROR`.

The lld_sqlstate() function

This function translates integer error codes into their corresponding SQL states.

Syntax

API

```
mi_string* lld_sqlstate (error);
   mi_integer                   error
```

ESQL/C

```
int* lld_sqlstate (int error);
```

error

An error code.

Return codes

On success, this function returns the SQL state value corresponding to the error code. On failure, returns `NULL`.

Important: This function returns a pointer to a constant, not to an allocated memory location.

Smart large object functions

The functions described in this section allow you to copy a smart large object to a file and to copy a smart large object to another smart large object. There is also a function that tells you whether the data in an lld_lob column is binary or character data.

The LOCopy function

This function creates a copy of a smart large object.

Syntax

SQL

```
CREATE FUNCTION LOCopy (lob LLD_Lob)
   RETURNS LLD_Lob ;

CREATE FUNCTION LOCopy (lob, LLD_Lob, table_name, CHAR(18),
column_name, CHAR(18))
   RETURNS LLD_Lob;
;
```

lob

A pointer to the smart large object to copy.

table_name

A table name. This parameter is optional.

column_name

A column name. This parameter is optional.

Usage

This function is an overloaded version of the LOCopy built-in server function. This function is identical to the built-in version of the function, except the first parameter is an lld_lob type rather than a BLOB or CLOB type.

The *table_name* and *column_name* parameters are optional. If you specify a *table_name* and *column_name*, LOCopy uses the storage characteristics from the specified *column_name* for the new smart large object that it creates.

If you omit *table_name* and *column_name*, LOCopy creates a smart large object with system-specified storage defaults.

See the description of the LOCopy function in the *HCL OneDB™ Guide to SQL: Syntax* for complete information about this function.

Return codes

This function returns a pointer to the new lld_lob value.

Context

LOCopy in the *HCL OneDB™ Guide to SQL: Syntax*.

The LOToFile function

Copies a smart large object to a file.

Syntax

SQL

```
CREATE FUNCTION LOToFile(lob LLD_Lob, pathname LVARCHAR,
file_dest CHAR(6)
   RETURNS LVARCHAR;
```

lob

A pointer to the smart large object.

pathname

A directory path and name of the file to create.

file_dest

The computer on which the file resides. Specify either `server` or `client`.

Usage

This function is an overloaded version of the LOToFile built-in server function. This function is identical to the built-in version of the function, except the first parameter is an lld_lob type rather than a BLOB or CLOB type.

See the description of the LOToFile function in the *HCL OneDB™ Guide to SQL: Syntax* for complete information about this function.

Return codes

This function returns the value of the new file name.

Context

LOToFile in the *HCL OneDB™ Guide to SQL: Syntax*.

The LLD_LobType function

Returns the type of data in an lld_lob column.

Syntax

SQL

```
CREATE FUNCTION LLD_LobType(lob LLD_Lob)
   RETURNS CHAR(4);
```

lob

A pointer to the smart large object

Usage

An lld_lob column can contain either binary or character data. You pass an lld_lob type to the LLD_LobType function to determine the type of data that the column contains.

Return codes

This function returns `blob` if the specified lld_lob contains binary data and `clob` if it contains character data.

Large Object Locator example code

This chapter provides example code that shows how to use some of the Large Object Locator functions together. It shows how to use all three of the Large Object Locator interfaces: SQL, server, and ESQL/C.

The SQL interface

The examples in this section show how to use the SQL interface to Large Object Locator.

The lld_lob type

The lld_lob is a user-defined type that you can use to specify the location of a smart large object and to specify whether the object contains binary or character data. The following subsections show how to use the lld_lob data type.

Implicit lld_lob casts

This section shows how to insert binary and character data into an lld_lob type column of a table. The following example makes use of implicit casts from BLOB and CLOB types to the lld_lob type.

Figure 1. Implicit lld_lob casts

```
create table slobs (key int primary key, slo lld_lob);

--Insert binary and text large objects into an lld_lob field
--Implicitly cast from blob/clob to lld_lob
insert into slobs values (1, filetoblob ('logo.gif', 'client'));

insert into slobs values (2, filetoclob ('quote1.txt', 'client'));

select * from slobs;


key  1
slo  blob:00608460a6b7c8d900000002000000030000000200000018000000000001000000608
     460736c6f000010029a2a6c92070000000000006c000af0cdd900000080006082500af0c9d
     e

key  2
slo  clob:00608460a6b7c8d900000002000000030000000300000019000000000001000000608
     460736c6f000010029a2a6c930d0000000000006c000af0cdd90000016000000010af0c9d
     e
```

The **slobs** table, created in this example, contains the **slo** column, which is of type lld_lob. The first INSERT statement uses the filetoblob function to copy a binary large object to a smart large object. There exists an implicit cast from a BLOB type to an lld_lob type, so the INSERT statement can insert the BLOB type large object into an lld_lob type column.

Likewise, there is an implicit cast from a CLOB type to an lld_lob type, so the second INSERT statement can insert a CLOB type large object into the **slo** column of the **slobs** table.

The SELECT statement returns the lld_lob types that identify the two smart large objects stored in the **slobs** table.

The **slo** column for key 1 contains an instance of an lld_lob type that identifies the data as BLOB data and contains a hexadecimal number that points to the location of the data.

The **slo** column for key 2 identifies the data as CLOB data and contains a hexadecimal number that points to the location of the data.

Explicit lld_lob casts

The example in the following figure shows how to select large objects of type BLOB and CLOB from a table and how to copy them to a file.

This example uses the **slobs** table created in Figure 1: Implicit lld_lob casts on page 35.

Figure 2. Explicit lld_lob casts

```
--Explicitly cast from lld_lob to blob/clob
select slo::blob from slobs where key = 1;


(expression)  <SBlob Data>


select slo::clob from slobs where key = 2;


(expression)
Ask not what your country can do for you,
but what you can do for your country.
```

The first SELECT statement retrieves the data in the **slo** column associated with key 1 and casts it as BLOB type data. The second SELECT statement retrieves the data in the **slo** column associated with key 2 and casts it as CLOB type data.

The LLD_LobType function

The following example shows how to use the LLD_LobType function to obtain the type of data—BLOB or CLOB—that an lld_lob column contains.

The **slobs** table in this example is the same one created in Figure 1: Implicit lld_lob casts on page 35. That example created the table and inserted a BLOB type large object for key 1 and a CLOB type large object for key 2.

Figure 3. The LLD_LobType function

```
-- LLD_LobType UDR
select key, lld_lobtype(slo) from slobs;

        key (expression)

          1 blob
          2 clob


select slo::clob from slobs where lld_lobtype(slo) = 'clob';


(expression)
Ask not what your country can do for you,
but what you can do for your country.
```

The first SELECT statement returns:

```
1 blob
2 clob
```

indicating that the data associated with key 1 is of type BLOB and the data associated with key 2 is of type CLOB.

The second SELECT statement uses LLD_LobType to retrieve the columns containing CLOB type data. The second SELECT statement casts the **slo** column (which is of type lld_lob) to retrieve CLOB type data.

The lld_locator type

The lld_locator type defines a large object. It identifies the type of large object and points to its location. It contains three fields:

lo_protocol

Identifies the kind of large object.

lo_pointer

A pointer to a smart large object or is `NULL` if the large object is any kind of large object other than a smart large object.

lo_location

A pointer to the large object, if it is not a smart large object. Set to `NULL` if it is a smart large object.

The examples in this section show how to:

Insert an lld_locator row into a table

The following example creates a table with an lld_locator row and shows how to insert a large object into the row.

Figure 4. Insert an lld_locator row into a table

```
--Create lobs table
create table lobs (key int primary key, lo lld_locator);

-- Create an lld_locator for an existing server file
insert into lobs
    values (1, "row('ifx_file',null,'/tmp/quote1.txt')");
```

The INSERT statement inserts an instance of an lld_locator row into the **lobs** table. The protocol in the first field, IFX_FILE, identifies the large object as a server file. The second field, *lo_pointer*, is used to point to a smart large object. Because the object is a server file, this field is `NULL`. The third field identifies the server file as `quote1.txt`.

Create a smart large object

The following example creates a smart large object containing CLOB type data. The lld_create function in figure creates a smart large object. The first parameter to lld_create uses the IFX_CLOB protocol to specify CLOB as the type of object to create. The other two arguments are `NULL`.

The lld_create function creates the CLOB type large object and returns an lld_locator row that identifies it.

The insert statement inserts in the **lobs** table the lld_locator row returned by lld_create.

Figure 5. Using lld_create

```
--Create a new clob using lld_create
insert into lobs
    values (2, lld_create ("row('ifx_clob',null,null)"::lld_locator));
```

Copy a client file to a large object

The following example uses the **lobs** table created in Figure 5: Using lld_create on page 37.

In the example, the lld_fromclient function in the first SELECT statement, copies the client file, `quote2.txt`, to an lld_locator row in the **lobs** table.

Figure 6. Copy a client file to a large object

```
-- Copy a client file to an lld_locator
select lld_fromclient ('quote2.txt', lo) from lobs where key = 2;


(expression)  ROW('IFX_CLOB      ','clob:ffffffffa6b7c8d9000000020000000300
          0000090000001a000000000001000000000000ad3c3dc000000000b06eec8000
          00000005c4e6000607fdc00000000000000000000000',NULL)


select lo.lo_pointer::clob from lobs where key = 2;


(expression)
To be or not to be,
that is the question.
```

The lld_fromclient function returns a pointer to the lld_locator row that identifies the data copied from the large object. The first SELECT statement returns this lld_locator row.

The next SELECT statement selects the *lo_pointer* field of the lld_locator row, `lo.lo_pointer`, and casts it to CLOB type data. The result is the data itself.

Copy a large object to a large object

The following example uses the **lobs** table created in Figure 4: Insert an lld_locator row into a table on page 37.

The lld_copy function in the example copies large object data from one lld_locator type row to another.

Figure 7. Copy a large object to a large object

```
-- Copy an lld_locator to an lld_locator
select lld_copy (S.lo, D.lo) from lobs S, lobs D where S.key = 1 and D.key = 2;


(expression)  ROW('IFX_CLOB      ','clob:ffffffffa6b7c8d9000000020000000300
          0000090000001a000000000001000000000000ad3c3dc000000000b06eec8000
          00000005c4e6000607fdc00000000000000000000000',NULL)


select lo.lo_pointer::clob from lobs where key = 2;


(expression)
Ask not what your country can do for you,
but what you can do for your country.
```

The second SELECT statement casts `lo.lo_pointer` to a CLOB type to display the data in the column.

Copy large object data to a client file

The following example uses the **lobs** table created in Figure 4: Insert an lld_locator row into a table on page 37. The lld_toclient function in Copy large object data to a client file on page 39 copies large object data to the `output.txt` client file. This function returns `t` when the function succeeds. The SELECT statement returns `t`, or `true`, indicating that the function returned successfully.

Figure 8. Copy large object data to a client file

```
-- Copy an lld_locator to a client file
select lld_toclient (lo, 'output.txt') from lobs where key = 2;

(expression)

           t
```

Create and delete a server file

The following example shows how to create a server file and then delete it.

The lld_copy function copies a large object to another large object. The lld_locator rows for the source and destination objects use the IFX_FILE protocol to specify a server file as the type of large object. The lld_copy function returns an lld_locator row that identifies the copy of the large object.

The INSERT statement inserts this row into the **lobs** table using 3 as the key.

Figure 9. Create and delete a server file

```
-- Create and delete a new server file
insert into lobs
    values (3, lld_copy (
        "row('ifx_file',null,'/tmp/quote2.txt')"::lld_locator,
        "row('ifx_file',null,'/tmp/tmp3')"::lld_locator));

select lo from lobs where key = 3;


lo  ROW('IFX_FILE          ',NULL,'/tmp/tmp3')


select lld_delete (lo) from lobs where key = 3;

(expression)

           t


delete from lobs where key = 3;
```

The first SELECT statement returns the lld_locator row identifying the large object.

The lld_delete function deletes the large object itself. The DELETE statement deletes the lld_locator row that referenced the large object.

The API interface

This section contains one example that shows how to use the Large Object Locator functions to create a user-defined routine. This routine copies part of a large object to another large object.

Create the lld_copy_subset function

The example shows the code for the lld_copy_subset user-defined routine. This routine copies a portion of a large object and appends it to another large object.

```
/* LLD SAPI interface example */

#include <mi.h>
#include <lldsapi.h>

/* append a (small) subset of a large object to another large object */

MI_ROW*
lld_copy_subset (MI_ROW* src,           /* source LLD_Locator */
                 MI_ROW* dest,          /* destination LLD_Locator */
                 mi_int8* offset,       /* offset to begin copy at */
                 mi_integer nbytes,     /* number of bytes to copy */
                 MI_FPARAM* fp)
{
    MI_ROW*        new_dest;       /* return value */
    MI_CONNECTION* conn;           /* database server connection */
    mi_string*     buffer;         /* I/O buffer */
    LLD_IO*        io;             /* open large object descriptor */
    mi_int8        new_offset;     /* offset after seek */
    mi_integer     bytes_read;     /* actual number of bytes copied */
    mi_integer     error;          /* error argument */
    mi_integer     _error;         /* extra error argument */
    mi_boolean     created_dest;   /* did we create the dest large object? */

    /* initialize variables */
    new_dest = NULL;
    conn = NULL;
    buffer = NULL;
    io = NULL;
    error = LLD_E_OK;
    created_dest = MI_FALSE;

    /* open a connection to the database server */
    conn = mi_open (NULL, NULL, NULL);
    if (conn == NULL)
        goto bad;

    /* allocate memory for I/O */
    buffer = mi_alloc (nbytes);
    if (buffer == NULL)
        goto bad;

    /* read from the source large object */
    io = lld_open (conn, src, LLD_RDONLY, &error);
    if (error != LLD_E_OK)
        goto bad;

    lld_seek (conn, io, offset, LLD_SEEK_SET, &new_offset, &error);
    if (error != LLD_E_OK)
        goto bad;
```

Figure 10. The lld_copy_subset function

```
    bytes_read = lld_read (conn, io, buffer, nbytes, &error);
    if (error != LLD_E_OK)
        goto bad;

    lld_close (conn, io, &error);
    if (error != LLD_E_OK)
        goto bad;

    /* write to the destination large object */
    new_dest = lld_create (conn, dest, &error);
    if (error == LLD_E_OK)
        created_dest = MI_TRUE;
    else if (error != LLD_E_EXISTS)
        goto bad;

    io = lld_open (conn, new_dest, LLD_WRONLY | LLD_APPEND | LLD_SEQ, &error);
    if (error != LLD_E_OK)
        goto bad;

    lld_write (conn, io, buffer, bytes_read, &error);
    if (error != LLD_E_OK)
        goto bad;

    lld_close (conn, io, &error);
    if (error != LLD_E_OK)
        goto bad;

    /* free memory */
    mi_free (buffer);

    /* close the database server connection */
    mi_close (conn);

    return new_dest;

    /* error clean up */
bad:
    if (io != NULL)
        lld_close (conn, io, &_error);
    if (created_dest)
        lld_delete (conn, new_dest, &_error);
    if (buffer != NULL)
        mi_free (buffer);
    if (conn != NULL)
        mi_close (conn);
    lld_error_raise (conn, error);
    mi_fp_setreturnisnull (fp, 0, MI_TRUE);
    return NULL;
}
```

The lld_copy_subset function defines four parameters:

- A source large object (lld_locator type)
- A destination large object (lld_locator type)

- The byte offset to begin copying
- The number of bytes to copy

It returns an lld_locator, identifying the object being appended.

The mi_open function opens a connection to the database. A buffer is allocated for I/O.

The following Large Object Locator functions are called for the source object:

lld_open

OpenS the source object

lld_seek

Seeks to the specified byte offset in the object

lld_read

Reads the specified number of bytes from the object

lld_close

Closes the object

The following Large Object Locator functions are called for the destination object:

- **lld_open**, to open the destination object
- **lld_write**, to write the bytes read from the source into the destination object
- **lld_close**, to close the destination object

The mi_close function closes the database connection.

This function also contains error-handling code. If the database connection cannot be made, if memory cannot be allocated, or if any of the Large Object Locator functions returns an error, the error code is invoked.

The error code handling code (`bad`) does one or more of the following actions, if necessary:

- Closes the source file
- Deletes the destination file
- Frees the buffer
- Closes the database connection
- Raises an error

You should establish a callback for exceptions (this example code, in the interest of simplicity and clarity, does not do so). See the *HCL OneDB™ DataBlade® API Programmer's Guide* for more information.

The lld_copy_subset routine

The following example shows how to use the lld_copy_subset user-defined routine defined in the previous section.

Figure 11. The lld_copy_subset routine

```
-- Using the lld_copy_subset function

create function lld_copy_subset (lld_locator, lld_locator, int8, int)
    returns lld_locator
    external name '/tmp/sapidemo.so'
    language c;

insert into lobs
    values (5, lld_copy_subset (
        "row('ifx_file',null,'/tmp/quote3.txt')"::lld_locator,
    "row('ifx_clob',null,null)"::lld_locator, 20, 70));

select lo from lobs where key = 5;
select lo.lo_pointer::clob from lobs where key = 5;
```

The lld_copy_subset function copies 70 bytes, beginning at offset 20 from the `quote3.txt` file, and appends them to a CLOB object. The INSERT statement inserts this data into the **lobs** table.

The first SELECT statement returns the lld_locator that identifies the newly copied CLOB data. The second SELECT statement returns the data itself.

Large Object Locator error handling

This chapter describes how to handle errors when calling Large Object Locator functions. It also lists and describes specific Large Object Locator errors.

There are two methods by which Large Object Locator returns errors to you:

- Through the error argument of a Large Object Locator function
- Through an exception

Both the API and ESQL/C versions of Large Object Locator functions use the error argument. Exceptions are returned only to the API functions.

Large Object Locator errors

All Large Object Locator functions use the return value to indicate failure. Functions that return a pointer return `NULL` in the event of failure. Functions that return an integer return `-1`.

Large Object Locator functions also provide an error code argument that you can test for specific errors. You can pass this error code to lld_error_raise()—which calls mi_db_error_raise if necessary to generate an MI_EXCEPTION—and propagate the error up the calling chain.

For ESQL/C functions, the LLD_E_SQL error indicates that an SQL error occurred. You can check the SQLSTATE variable to determine the nature of the error.

When an error occurs, Large Object Locator functions attempt to reclaim any outstanding resources. You should close any open large objects and delete any objects you have created that have not been inserted into a table.

A user-defined routine that directly or indirectly calls a Large Object Locator function (API version) can register a callback function. If this function catches and handles an exception and returns control to the Large Object Locator function, Large Object Locator returns the LLD_E_EXCEPTION error. You can handle this error as you would any other: close open objects and delete objects not inserted in a table.

Error handling exceptions

You should register a callback function to catch exceptions generated by underlying DataBlade® API functions called by Large Object Locator functions. For example, if you call lld_read() to open a smart large object, Large Object Locator calls the DataBlade® API mi_lo_read() function. If this function returns an error and generates an exception, you must catch the exception and close the object you have open for reading.

Use the mi_register_callback() function to register your callback function. The callback function should track all open large objects, and in the event of an exception, close them. You can track open large objects by creating a data structure with pointers to **LLD_IO** structures, the structure that the lld_open() function returns when it opens an object. Use the lld_close() function to close open large objects.

Error codes

This section lists and describes the Large Object Locator error codes.

Error code	SQL state	Description
LLD_E_INTERNAL	ULLD0	Internal Large Object Locator error. If you receive this error, call HCL® OneDB® Technical Support.
LLD_E_OK	N.A.	No error.
LLD_E_EXCEPTION	N.A.	MI_EXCEPTION raised and handled. Applies to API only.
LLD_E_SQL	N.A.	SQL error code in SQLSTATE/SQLCODE. Applies to ESQL/C interface only.
LLD_E_ERRNO	ULLD1	OS (UNIX/POSIX)
LLD_E_ROW	ULLD2	Passed an invalid MI_ROW type. The type should be lld_locator. This is an API error only.
LLD_E_PROTOCOL	ULLD3	Passed an invalid or unsupported *lo_protocol* value.
LLD_E_LOCATION	ULLD4	Passed an invalid *lo_location* value.
LLD_E_EXISTS	ULLD5	Attempted to (re)create an existing large object.
LLD_E_NOTEXIST	ULLD6	Attempted to open a nonexistent large object.
LLD_E_FLAGS	ULLD7	Used invalid flag combination when opening a large object.
LLD_E_LLDIO	ULLD8	Passed a corrupted **LLD_IO** structure.

Error code	SQL state	Description
LLD_E_RDONLY	ULLD9	Attempted to write to a large object that is open for read-only access.
LLD_E_WRONLY	ULLDA	Attempted to read from a large object that is open for write-only access.
LLD_E_SEQ	ULLDB	Attempted to seek in a large object that is open for sequential access only.
LLD_E_WHENCE	ULLDC	Invalid whence (seek) value.
LLD_E_OFFSET	ULLDD	Attempted to seek to an invalid offset.
N.A.	ULLDO	Specified an invalid lld_lob input string.
N.A.	ULLDP	Specified an invalid lld_lob type.
N.A.	ULLDQ	Attempted an invalid cast of an lld_lobtype into a BLOB or CLOB type.
N.A.	ULLDR	Used an invalid import file specification with the lld_lob type.

MQ Messaging

(WMQ) messaging products provide an infrastructure for distributed, asynchronous communication of data in a distributed, heterogeneous environment. The WMQ message queue allows you to easily exchange information across platforms.

The MQ extension provides the functionality to exchange messages between databases and WMQ message queues.

You can replicate MQ messages with all types of high-availability clusters. If a secondary server in a cluster is read-only, the non-WMQ data cannot be updated from that server, however the WMQ message data can be updated.

About MQ messaging

You can use either functions or tables to communicate between a database server application and the queue.

Applications can send and receive messages from local or remote queue managers that reside anywhere in the network and participate in a transaction. There is no limit to the number of queue managers that can participate in a transaction.

WMQ platform requirements are independent of your database server platform requirements. For more information about respective platform requirements, see the WMQ documentation and your machine notes.

Prepare to use MQ messaging

Before you can use MQ messaging, you must install and configure (WMQ) and configure your database server for use with WMQ.

The database server comes with a server-based messaging library and a client-based messaging library. The server-based messaging library is default option.

To use MQ messaging, you perform these tasks:

1. Decide whether to use the server-based MQ messaging or client-based messaging library.
2. Install WMQ HCL OneDB™ and set up the queue manager, queues, and channels.

   When you use the server-based messaging library, the database server connects to the queue manager that resides on the same computer. Therefore, you must install HCL OneDB™ and the WMQ Server on the same computer.

   When you use the client-based messaging library, the database server uses a network protocol to connect to the queue manager anywhere on the network. You must install the database server and the WMQ Client on the same computer. You can install the WMQ server on the same computer or on different computers on the network. If you plan to use local queue managers, you must install the database server and WMQ on the same computer. See WebSphere® MQ documentation for installation details.
3. Verify that MQ messaging is working correctly.
4. Use MQ functions or tables in your application.

If you configure your system to use both server-based and client-based MQ messaging on your database server, you can switch between the two methods of messaging. You cannot use both methods at the same time on a database server instance

Install and configure WMQ

You must install and configure before using MQ messaging.

Information about how to install WMQ is included in the WMQ product documentation.

A WMQ queue manager is a system program that provides queuing services to applications. It provides an application programming interface for programs to access messages on the queues managed by a WMQ message broker. Applications can send and receive messages to and from a queue.

As necessary, you need to complete the following WMQ queue configuration:

- Create a queue manager.
- Create a queue.
- Create a subscriber queue.

For instructions on how to create a queue manager, a queue, and a subscriber queue, see the platform-specific documentation received with your WMQ product.

Prepare your database server for MQ messaging

You must prepare your HCL OneDB™ database for MQ messaging.

To prepare for MQ messaging, add user **informix** to the **mqm** group and restart the database server. Only members of the **mqm** group are authorized to access to WMQ queues. For more information, see the platform-specific documentation for WMQ.

The **mq** virtual processor is created automatically the first time you access an MQ messaging table or run an MQ messaging function.

The HCL OneDB™ database server has the following additional requirements:

- Non-logged databases are not supported.
- ANSI databases are not supported.
- The Scheduler must be running.

If you attempt to run an MQ messaging function in an unlogged or ANSI database, a message that `DataBlade registration failed` is printed in the online message log. If the Scheduler is not running the first time that you access an MQ messaging table or run an MQ messaging function, a message that the table cannot be found or the routine cannot be resolved is returned.

Sample code for setting up queue managers, queues, and channels

After you install either the (WMQ) server or both the WMQ server and client, you can set up the queue manager, queues, and channels.

You must only set up channels if you plan to use a WMQ client-based library. For information about channels, see your documentation.

The following example shows how to set up the queue manager, queues, and channels:

1. Create queue manager `lqm1`, using `-q` to specify the default queue manager:
   ```
   crtmqm –q lqm1
   ```
2. Start the queue manager:
   ```
   strmqm lqm1
   ```
3. Start the publish/subscribe service:
   ```
   strmqbrk –m lqm1
   ```
4. Stop the queue manager:
   ```
   endmqm –w lqm1
   ```
5. Delete the queue manager:
   ```
   dltmqm lqm1
   ```
6. Start the TCP listener on port 1414 for queue manager lqm1:
   ```
   runmqlsr –t tcp –m lqm1 –p 1414 &
   ```
7. Run the following commands in runmqsc lqm1:
   ```
   DEFINE CHANNEL(QM1CH) CHLTYPE(SVRCONN) TRPTYPE(TCP) +
   DESCR('Server connection to WMQ client') REPLACE

   DEFINE CHANNEL(QM1CH) CHLTYPE(CLNTCONN) TRPTYPE(TCP) +
   CONNAME('hostname(1414)') +
   ```

```
DESCR('WebSphere MQ client connection to server 1') +
QMNAME('lqm1') REPLACE
```

8. Create database server-related queues by running the following command:

```
runmqsc lqm1 < $ONEDB_HOME/extend/mqblade.2.0/idsdefault.tst
```

9. Copy AMQCLCHL.TAB to the WMQ default location.

Sample code for setting up the server for use with WMQ

After you install either the IBMWebSphere MQ (WMQ) server or both the WMQ server and client, you can set up the database server for use with WMQ.

The following example shows how to set up the database server for MQ:

1. Open DB-Access and the **stores_demo** database.
2. Run the following commands:

```
-- Service for most operations

INSERT INTO mqiservice
         (servicename, queuemanager, queuename)
  VALUES ('lser.qm1', 'lqm1', 'IDS.DEFAULT.QUEUE');

-- Service for publishing

INSERT INTO mqiservice
         (servicename, queuemanager, queuename)
  VALUES ('lpubser.qm1', 'lqm1', 'SYSTEM.BROKER.DEFAULT.STREAM');

-- service for subscribing
INSERT INTO mqiservice
         (servicename, queuemanager, queuename, mqchllib, mqchltab)
  VALUES ('lsubser.qm1', 'lqm1', 'SYSTEM.BROKER.CONTROL.QUEUE');

-- service for receiving subscribe message
INSERT INTO mqiservice
         (servicename, queuemanager, queuename, mqchllib, mqchltab)
VALUES ('lrecsubser.qm1', 'lqm1',
 'IDS.DEFAULT.SUBSCRIBER.RECEIVER.QUEUE');

-- subscriber information
INSERT INTO mqipubsub
         (pubsubname, servicebroker, receiver,psstream,pubsubtype)
  VALUES ('lsub.qm1', 'lsubser.qm1', 'lrecsubser.qm1',
          'SYSTEM.BROKER.DEFAULT.STREAM', 'Subscriber');

-- publisher information
INSERT INTO mqipubsub
         (pubsubname, servicebroker, receiver,psstream, pubsubtype)
  VALUES ('lpub.qm1', 'lpubser.qm1', '', '', 'Publisher');
```

Switch between server-based and client-based messaging

If you are set up to use both server-based and client-based MQ messaging on your database server, you can switch between the two methods of messaging. Server-based messaging is the default method.

The commands you use for switching to server-based messaging and switching to client-based messaging, which are described in the subtopics below, differ slightly.

Switching from server-based to client-based messaging

You can switch from server-based messaging, the default method for messaging, to client-based messaging.

Before you begin

Prerequisites:

- When you switch to client-based messaging, the database server and (WMQ) must be installed on the same computer.
- On Windows™, you must have the MKS Toolkit to run the chown command.

About this task

To switch to server-based messaging:

1. Bring down the database server.
2. Run this command: cd $ONEDB_HOME/extend/mqblade.2.0
3. Run this command: rm idsmq.bld
4. Run either of the following commands:
   Choose from:
   - cp idsmqc.bld idsmq.bld
   - ln –s idsmqc.bld idsmq.bld

   Note that these commands differ slightly from the commands used to switch to server-based messaging.
5. Run this command: chown Informix:Informix idsmq.bld
6. Run this command: chmod –w idsmq.bld
7. Start the database server.

Switching from client-based to server-based messaging

If you previously switched to client-based messaging, you can switch back to server-based messaging.

Before you begin

Prerequisites:

- When you switch to server-based messaging, the database server and (WMQ) can be present on the same computer or on a different computer on the network.
- On Windows™, you must have the MKS Toolkit to run the chown command.

About this task

To switch from client-based messaging to server-based messaging:

1. Bring down the database server.
2. Run this command: cd $ONEDB_HOME/extend/mqblade.2.0
3. Run this command: rm idsmq.bld
4. Run either of the following commands:

   Choose from:

   - cp idsmqs.bld idsmq.bld
   - ln –s idsmqs.bld idsmq.bld

   Note that these commands differ slightly from the commands used to switch to client-based messaging.
5. Run this command: chown Informix:Informix idsmq.bld
6. Run this command: chmod –w idsmq.bld
7. Start the database server.

Verification

After completely the necessary configuration, verify that MQ messaging is working correctly.

MQ functions must be used within a transaction. For functions that use the EXECUTE statement, you must explicitly start the transaction with a BEGIN WORK statement. For functions that use the SELECT, UPDATE, DELETE, or INSERT statements, you do not need to use a BEGIN WORK statement.

For more information about all of the functions used below, see MQ messaging functions on page 57.

Insert data into a queue

The service IDS.DEFAULT.SERVICE specifies the IDS.DEFAULT.QUEUE. Before inserting data into the queue, you should check the size of the queue.

After inserting the data, you should check the queue to confirm that the data was added.

```
BEGIN WORK;

EXECUTE FUNCTION MQSend('IDS.DEFAULT.SERVICE', 'IDS.DEFAULT.POLICY', 'hello queue');

(expression)            1
1 row(s) retrieved.

COMMIT WORK;
```

Read an entry from a queue

The MQRead() function reads a message from the queue but does not remove it.

After reading the message, the queue has not been changed:

```
BEGIN WORK;
```

```
EXECUTE FUNCTION MQRead('IDS.DEFAULT.SERVICE', 'IDS.DEFAULT.POLICY');

(expression) hello queue

1 row(s) retrieved.

COMMIT WORK;
```

The following example reads a message from the queue and inserts it into a database table:

```
INSERT into msgtable values (MQRead('IDS.DEFAULT.SERVICE', 'IDS.DEFAULT.POLICY'));

1 row(s) inserted.

SELECT * from msgtable;

msg hello queue

1 row(s) retrieved.

COMMIT WORK;
```

Receive an entry from a queue

The MQReceive() function removes the message from the queue.

The following example shows the removal of message from the queue:

```
BEGIN WORK;

EXECUTE FUNCTION MQReceive('IDS.DEFAULT.SERVICE', 'IDS.DEFAULT.POLICY');

(expression)  hello queue

1 row(s) retrieved.

COMMIT WORK;
```

Publish and subscribe to a queue

Publishing and subscribing to a queue is an effective way of exchanging information between multiple users.

MQ messaging interacts directly with the WMQ Publish/Subscribe component. The component allows a message to be sent to multiple subscribers based on a topic. Users subscribe to a topic, and when a publisher inserts a message with that topic into the queue, the WMQ broker routes the messages to all of the queues of each specified subscriber. Then, the subscriber retrieves the message from the queue.

Subscribe to a queue

To subscribe to a queue, use the MQSubscribe() function.

The following example shows how a database application subscribes to a queue to receive messages for a topic named *"Weather"*:

```
   --- before subscribe
   Topic: MQ/TIMESERIES.QUEUE.MANAGER           /StreamSupport
   Topic: MQ/S/TIMESERIES.QUEUE.MANAGER         /Subscribers/Identities/*
   Topic: MQ/S/TIMESERIES.QUEUE.MANAGER         /Subscribers/Identities/*
   Topic: MQ/S/TIMESERIES.QUEUE.MANAGER         /Subscribers/Identities/*

BEGIN WORK;

EXECUTE FUNCTION MQSubscribe('AMT.SAMPLE.SUBSCRIBER', 'AMT.SAMPLE.PUB.SUB.POLICY',
'Weather');

(expression)           1

1 row(s) retrieved.

   --- after subscribe
   Topic: MQ/TIMESERIES.QUEUE.MANAGER           /StreamSupport
   Topic: MQ/S/TIMESERIES.QUEUE.MANAGER         /Subscribers/Identities/*
   Topic: Weather
   Topic: MQ/S/TIMESERIES.QUEUE.MANAGER         /Subscribers/Identities/*
   Topic: MQ/S/TIMESERIES.QUEUE.MANAGER         /Subscribers/Identities/*

COMMIT WORK;
```

Unsubscribe from a queue

To unsubscribe from a queue, use the MQUnsubscribe() function.

For example, specify:

```
BEGIN WORK;

EXECUTE FUNCTION MQUnsubscribe('AMT.SAMPLE.SUBSCRIBER', 'AMT.SAMPLE.PUB.SUB.POLICY',
'Weather');(

expression)           1

1 row(s) retrieved.
   Topic: MQ/TIMESERIES.QUEUE.MANAGER             /StreamSupport
   Topic: MQ/S/TIMESERIES.QUEUE.MANAGER           /Subscribers/Identities/*
   Topic: MQ/S/TIMESERIES.QUEUE.MANAGER           /Subscribers/Identities/*
   Topic: MQ/S/TIMESERIES.QUEUE.MANAGER           /Subscribers/Identities/*

COMMIT WORK;
```

Publish to a queue

To publish to a queue, use the MQPublish() function.

For example, specify:

```
BEGIN WORK;

EXECUTE FUNCTION MQPublish('IDS.DEFAULT.SERVICE', 'IDS.DEFAULT.POLICY', 'Weather');

(expression)           1
```

```
COMMIT WORK;
```

MQ messaging tables

You use Virtual-Table Interface (VTI) access method to access WMQ queues using table semantics.

VTI binds tables to WMQ queues, creating transparent access to WMQ objects and enabling users to access the queue as if it were a table. For more information about VTI, see the *HCL OneDB™ Virtual-Table Interface Programmer's Guide*.

Schema mapping

When a table is bound to a WMQ queue, the schema is mapped directly to WMQ objects.

The following table shows the mapping of schema to WMQ objects.

Table 2. Schema mapping to WMQ objects

Name	Type	Description
msg	lvarchar(*maxMessage*)	The message being sent or received. The default size is 4,000; the limit is 32,628.
correlid	varchar(24)	The correlation ID, which can be used as a qualifier
topic	varchar(40)	The topic used with publisher or subscriber, which can be used as a qualifier
qname	varchar(48)	The name of the queue
msgid	varchar(12)	The message ID
msgformat	varchar(8)	The message format

General table behavior

WMQ metadata tables operate in specified ways.

For every table created, the following applies:

- The PUBLIC group is limited to SELECT privileges. Only the database administrator and the table creator have INSERT privileges.
- When a function is first invoked in each user session, WMQ metadata tables are read and their values are cached in the PER_SESSION memory. The cache is not refreshed until the session closes or the database is closed and reopened.

Create and bind a table

Use the MQCreateVtiReceive() function to create a table and bind it to a queue.

The following example creates a table named **vtimq**, and binds it to the queue defined by service **IDS.DEFAULT.SERVICE** and policy **IDS.DEFAULT.POLICY**.

```
BEGIN WORK;

EXECUTE FUNCTION MQICreateVtiReceive ("VtiMQ",
                "IDS.DEFAULT.SERVICE",  "IDS.DEFAULT.POLICY");
```

Using a SELECT statement on a table created with MQCreateVtiReceive(), results in a message is received from the table, which is the equivalent of calling the MQReceive() function on the queue. For both functions, the messages selected are removed from the queue.

To browse the messages on the queue without removing the messages from the queue, use the MQCreateVtiRead() function. In the following example, MQCreateVtiRead() binds the table **vtimq** to a queue:

```
BEGIN WORK;

EXECUTE FUNCTION MQCreateVtiRead (vtimq, read-service, policy, maxMessage)
```

For complete information about the MQCreateVtiRead() or MQCreateVtiReceive() functions, see MQ messaging functions on page 57.

Use INSERT and SELECT

After a table is bound to a queue, use INSERT to insert items into the WMQ queue, and SELECT to retrieve WMQ messages.

Using the example with table **vtimq** above, the following example inserts a message into the **msg** column of **VtiMQ** and into the queue described by IDS.DEFAULT.SERVICE service and policy IDS.DEFAULT.POLICY:

```
INSERT into VtiMQ (msg) values ('PUT on queue with SQL INSERT');
1 row(s) inserted.
```

Use a SELECT statement to display the message:

```
SELECT * from VtiMQ;
msg        PUT on queue with SQL INSERT
correlid
topic
qname      IDS.DEFAULT.QUEUE
msgid      AMQ
msgformat  MQSTR
```

Retrieve the queue element

Use the MQRead() function to retrieve the queue element.

For example:

```
BEGIN WORK;

EXECUTE FUNCTION MQRead('IDS.DEFAULT.SERVICE', 'IDS.DEFAULT.POLICY');
(expression)  PUT on queue with SQL INSERT
1 row(s) retrieved.
COMMIT WORK
```

Special considerations

Binding a table to a queue creates a useful interface between the queue and the database. However, due to the inherent limitations of a queue, not all database functionality can be used.

When a message is fetched from a queue, the default database processing is to dequeue, or remove, it. Every time a queue is read by the database, the data within the queue changes. This behavior differs from a standard read by a database, in which the data does not change. Supplying only a mapping that enables users to browse, where reading does not remove the queue, eliminates a major queue functionality. Enabling both processing models provides more options and requires corresponding responsibility.

By default, the top element is removed when a message is fetched from a queue. WMQ allows messages to be retrieved based upon a *correlid*. A correlid is a correlation identifier that can be used as a key, for example, to correlate a response message to a request message. If the correlid of the message matches the correlid of a request, the message is returned. If the VTI table is qualified with the correlid column, the correlid qualifier is passed into the WMQ request to fetch a value.

In the following example, a queue has three messages and only the second message contains a correlid, which is named **'fred'**. The following statement removes all three messages from the queue and places them in a table named **flounder**:

```
INSERT into flounder (deQueuedMsg) values (SELECT msg from vtimq);
```

When execution completes, no messages remain on the queue and three new rows appear in the **flounder** table.

The following example qualifies the **vtimq** table:

```
INSERT into flounder (deQueuedMsg) values (SELECT msg from vtimq where
correlid = 'fred');
```

The above statement creates two groups of messages:

- Messages that failed the ***correlid* = 'fred'** qualification
- Messages that passed the ***correlid* = 'fred'** qualification. The one message that passed the qualification is located in the **flounder** table.

Statements including qualifiers other than equality (=) or NULL return an error. Statements including NULL return unexpected results.

Table errors

Tables that are mapped to WMQ can generate non-database errors if the underlying WMQ request fails.

In the example below, a VTI mapping was established using a bad service definition, and the error was not recognized until a SELECT statement was executed against the table.

```
BEGIN WORK;
EXECUTE FUNCTION MQCreateVtiReceive('vtiTable',"BAD.SERVICE");
SELECT * from vtitable;

(MQ015) - FUNCTION:MqiGetServicePolicy, SERVICE:BAD.SERVICE,
POLICY:IDS.DEFAULT.POLICY ::
BAD.SERVICE is not present in the database "informix".MQISERVICE table.
```

```
Error in line 1
Near character position 23
```

MQ messaging functions

MQ messaging functions to enable applications to exchange data directly between the application and WebSphere® MQ.

All MQ messaging functions are created with a stack size of 64K. These MQ messaging functions can be executed within SQL statements and should have an explicit or implicit transactional context.

All MQ messaging functions or MQ messaging-based VTI tables can be invoked only on local (sub-ordinator) servers. Using MQ messaging functions or MQ messaging-based VTI tables on a remote server will return an error. MQ messaging functions cannot be used when HCL OneDB™is participating as a resource manager in an externally-managed global XA transaction.

MQ messaging functions use the **"informix".mqi*** service and policy tables to provide default values if the optional *policy* and *service* parameters are not specified.

Service and policy tables

MQ messaging functions use three service and policy tables.

Most of the MQ messaging functions have an optional *policy* and *service* parameter. If the parameter is not passed, the default value is used. The following table lists the default values for these parameters.

Table 3. Default policy and service values

Type	Name	Resources	Status
Service	IDS.DEFAULT.SERVICE	IDS.DEFAULT.QUEUE	created
Service	IDS.DEFAULT.SUBSCRIBER	SYSTEM.BROKER.CONTROL.QUEUE	system default
Service	IDS.DEFAULT.PUBLISHER	SYSTEM.BROKER.DEFAULT.STREAM	system default
Service	IDS.DEFAULT.SUBSCRIBER.RECEIVER	IDS.DEFAULT.SUBSCRIBER.RECEIVER.QUEUE	created
Policy	IDS.DEFAULT.POLICY	*connection name* :*default queuemanager*	system default
Publisher	IDS.DEFAULT.PUBLISHER	sender:IDS.DEFAULT.PUBLISHER	system default
Subscriber	IDS.DEFAULT.SUBSCRIBER	sender:IDS.DEFAULT.SUBSCRIBER receiver: IDS.DEFAULT.SUBSCRIBER.RECEIVER	system default

Each service definition includes a queue specification. The service can be mapped any queue. For testing purposes, you can create the following queues using the script `idsdefault.tst`:

- IDS.DEFAULT.QUEUE queue for the IDS.DEFAULT.SERVICE
- IDS.DEFAULT.SUBSCRIBER.RECIVER.QUEUE queue for the IDS.DEFAULT.SUBSCRIBER

The script `idsdefault.tst` is located in the `MQBLADE` directory. Use the runmqsc utility to execute commands in `idsdefault.tst`.

If the QueueManager is not a default queue manager, you must update the **queuemanager** column of the **informix.mqiservice** table by updating **servicename** to IDS.DEFAULT.SERVICE, IDS.DEFAULT.PUBLISHER, IDS.DEFAULT.SUBSCRIBER and IDS.DEFAULT.SUBSCRIBER.RECEIVER.

During registration, the following default values are inserted into the "**informix**".**mqi*** tables:

```
INSERT INTO ""informix"".mqiservice(servicename, queuemanager, queuename)
       VALUES('IDS.DEFAULT.SERVICE', '', 'IDS.DEFAULT.QUEUE');

INSERT INTO ""informix"".mqiservice(servicename, queuemanager, queuename)
       VALUES('IDS.DEFAULT.PUBLISHER', '', 'SYSTEM.BROKER.DEFAULT.STREAM');

INSERT INTO ""informix"".mqiservice(servicename, queuemanager, queuename)
       VALUES('IDS.DEFAULT.SUBSCRIBER', '', 'SYSTEM.BROKER.CONTROL.QUEUE');

INSERT INTO ""informix"".mqiservice(servicename, queuemanager, queuename)
       VALUES('IDS.DEFAULT.SUBSCRIBER.RECEIVER', '',
              'IDS.DEFAULT.SUBSCRIBER.RECEIVER.QUEUE');

INSERT INTO ""informix"".mqipubsub(pubsubname, servicebroker, receiver,
                                   psstream, pubsubtype)
        VALUES('IDS.DEFAULT.SUBSCRIBER', 'IDS.DEFAULT.SUBSCRIBER',
               'IDS.DEFAULT.SUBSCRIBER.RECEIVER',
               'SYSTEM.BROKER.DEFAULT.STREAM', 'Subscriber');

INSERT INTO ""informix"".mqipubsub(pubsubname, servicebroker, receiver,
                                   psstream, pubsubtype)
       VALUES('IDS.DEFAULT.PUBLISHER', 'IDS.DEFAULT.PUBLISHER', '', '',
              'Publisher');

INSERT INTO ""informix"".mqipolicy(policyname)
       VALUES('IDS.DEFAULT.POLICY');

INSERT INTO ""informix"".mqipolicy(policyname)
       VALUES('IDS.DEFAULT.PUB.SUB.POLICY');
```

The "informix".mqiservice table

The "**informix**".**mqiservice** table contains the service definitions for service point (sender/receiver) attributes.

The "**informix**".**mqiservice** table has the following schema:

```
CREATE TABLE "informix".mqiservice
    servicename     LVARCHAR(256),
    queuemanager    VARCHAR(48) NOT NULL,
    queuename       VARCHAR(48) NOT NULL,
    defaultformat   VARCHAR(8) default ' ',
    ccsid           VARCHAR(6) default ' ',
    mqconnname      lvarchar(264) default '',
```

```
    mqchannelname  varchar(20) default 'SYSTEM.DEF.SVRCONN',
    mqxpt           INTEGER DEFAULT 2  CHECK ( mqxpt >= 0 AND mqxpt <= 6 ),
    mqchllib       lvarchar(512) default '',
    mqchltab        lvarchar(512) default '',
    mqserver        lvarchar(512) default '',
    PRIMARY KEY (servicename) );
```

The attributes are defined as follows:

servicename

The service name used in the MQ functions.

queuemanager

The queue manager service provider.

queuename

The queue name to send the message to or receive the message from.

defaultformat

Defines the default format.

ccsid

The coded character set identifier of the destination application.

mqconnname

The MQ connection name. This value, which is present only when the client-based messaging library is used, enables the client application to connect to multiple server queue managers simultaneously.

mqchannelname

The MQ channel name. This value, which is present only when the client-based messaging library is used, enables the client application to connect to multiple server queue managers simultaneously.

mqxpt

The MQ transport type attribute. This value, which is present only when the client-based messaging library is used, enables the client application to connect to multiple server queue managers simultaneously.

mqchllib

The MQCHLLIB environment variable of WMQ. This value, which is present only when the client-based messaging library is used, specifies the directory path to the file containing the client channel definition table.

mqchltab

The MQCHLTAB environment variable of WMQ. This value, which is present only when the client-based messaging library is used, specifies the name of the file containing the client channel definition table

mqserver

The MQSERVER environment variable of WMQ. This value, which is present only when the client-based messaging library is used, defines a channel and specifies the location of the WebSphere® MQ server and the communication method that is used.

An application can specify the `mqchannelname`, `mqxpt`, and `mqconnname` attributes of a channel at run time. This enables the client application to connect to multiple server queue managers simultaneously. If these values are present, they take precedence over other values. For more information, see information about using the MQCNO structure on an MQCONNX call in the documentation.

Whenever each service is connected to WMQ, the service uses environment variables in the following order:

1. MQCNO values
2. Variables in the service
3. Variables in the instance
4. WMQ default values

The "informix".mqipubsub table

The **"informix".mqipubsub** table contains publisher definitions.

The **"informix".mqipubsub** table has the policy definitions for the following attributes:

- Distribution list
- Receive
- Subscriber
- Publisher

The **"informix".mqipubsub** table has the following schema:

```
CREATE TABLE "informix".mqipubsub
    pubsubname      LVARCHAR(256) NOT NULL UNIQUE,
    servicebroker   LVARCHAR(256),
    receiver        LVARCHAR(256) default ' ',
    psstream        LVARCHAR(256) default ' ',
    pubsubtype      VARCHAR(20) CHECK (pubsubtype IN ('Publisher', 'Subscriber')),
FOREIGN KEY (servicebroker) REFERENCES "informix".mqiservice(servicename));
```

The attributes are defined as follows:

pubsubname

is the name of the publish/subscribe service.

servicebroker

The service name of the publish/subscribe service.

receiver

The queue on which to receive messages after subscription.

psstream

The stream coordinating the publish/subscribe service.

pubsubtype

The service type.

The "informix".mqipolicy table

The "**informix**".**mqipolicy** table contains policy definitions.

The "**informix**".**mqipolicy** table has the policy definitions for the following attributes:

- General
- Publish
- Receive
- Reply
- Send
- Subscribe

The "**informix**".**mqipolicy** table has the following schema:

```
CREATE TABLE "informix".mqipolicy
    policyname           VARCHAR(128) NOT NULL,
    messagetype           CHAR(1) DEFAULT 'D' CHECK (messagetype IN ('D', 'R')),
    messagecontext       CHAR(1) DEFAULT 'Q' CHECK (messagecontext IN
                              ('Q','P','A','N')),
    snd_priority         CHAR(1) DEFAULT 'T' CHECK (snd_priority IN
                              ('0','1','2','3','4', '5','6','7','8','9', 'T')),
    snd_persistence      CHAR(1) DEFAULT 'T' CHECK (snd_persistence IN
                              ('Y','N','T')),
    snd_expiry           INTEGER DEFAULT -1 CHECK ( snd_expiry > 0 OR snd_expiry
                               = -1 ),
    snd_retrycount       INTEGER DEFAULT 0 CHECK ( snd_retrycount >= 0 ),
    snd_retry_intrvl     INTEGER DEFAULT 1000 CHECK ( snd_retry_intrvl >= 0 ),
    snd_newcorrelid      CHAR(1) DEFAULT 'N' CHECK ( snd_newcorrelid IN ('Y','N')),
    snd_resp_correlid    CHAR(1) DEFAULT 'M' CHECK ( snd_resp_correlid IN ('M','C')),
    snd_xcption_action   CHAR(1) DEFAULT 'Q' CHECK ( snd_xcption_action IN
                              ('Q','D')),
    snd_report_data      CHAR(1) DEFAULT 'R' CHECK ( snd_report_data IN
                              ('R','D','F')),
    snd_rt_exception     CHAR(1) DEFAULT 'N' CHECK ( snd_rt_exception IN ('Y','N')),
    snd_rt_coa           CHAR(1) DEFAULT 'N', CHECK ( snd_rt_coa IN ('Y','N')),
    snd_rt_cod           CHAR(1) DEFAULT 'N' CHECK ( snd_rt_cod IN ('Y','N')),
   snd_rt_expiry        CHAR(1) DEFAULT 'N' CHECK ( snd_rt_expiry IN ('Y','N')),
   reply_q               VARCHAR(48) DEFAULT 'SAME AS INPUT_Q',
    reply_qmgr           VARCHAR(48) DEFAULT 'SAME AS INPUT_QMGR',
    rcv_truncatedmsg     CHAR(1) DEFAULT 'N' CHECK ( rcv_truncatedmsg IN ('Y','N')),
    rcv_convert          CHAR(1) DEFAULT 'Y' CHECK ( rcv_convert IN ('Y','N')),
    rcv_poisonmsg        CHAR(1) DEFAULT 'N' CHECK ( rcv_poisonmsg IN ('Y','N')),
    rcv_openshared       CHAR(1) DEFAULT 'Q' CHECK ( rcv_openshared IN
                              ('Y','N','Q')),
    rcv_wait_intrvl      INTEGER DEFAULT 0 CHECK ( rcv_wait_intrvl >= -1 ),
   pub_suppressreg      CHAR(1) DEFAULT 'Y' CHECK ( pub_suppressreg IN ('Y','N')),
    pub_anonymous         CHAR(1) DEFAULT 'N' CHECK ( pub_anonymous IN ('Y','N')),
    pub_publocal         CHAR(1) DEFAULT 'N' CHECK ( pub_publocal IN ('Y','N')),
    pub_direct           CHAR(1) DEFAULT 'N' CHECK ( pub_direct IN ('Y','N')),
   pub_correlasid       CHAR(1) DEFAULT 'N' CHECK ( pub_correlasid IN ('Y','N')),
   pub_retain            CHAR(1) DEFAULT 'N' CHECK ( pub_retain IN ('Y','N')),
   pub_othersonly       CHAR(1) DEFAULT 'N' CHECK ( pub_othersonly IN ('Y','N')),
  sub_anonymous         CHAR(1) DEFAULT 'N' CHECK ( sub_anonymous IN ('Y','N')),
   sub_sublocal         CHAR(1) DEFAULT 'N' CHECK ( sub_sublocal IN ('Y','N')),
```

```
   sub_newpubsonly      CHAR(1) DEFAULT 'N' CHECK ( sub_newpubsonly IN ('Y','N')),
    sub_pubonreqonly    CHAR(1) DEFAULT 'N' CHECK ( sub_pubonreqonly IN ('Y','N')),
    sub_correlasid      CHAR(1) DEFAULT 'N' CHECK ( sub_correlasid IN ('Y','N')),
    sub_informifret     CHAR(1) DEFAULT 'Y' CHECK ( sub_informifret IN ('Y','N')),
    sub_unsuball        CHAR(1) DEFAULT 'N' CHECK ( sub_unsuball IN ('Y','N')),
    syncpoint           CHAR(1) DEFAULT 'Y' CHECK ( syncpoint IN ('Y','N'))
     PRIMARY KEY (policyname) );
```

The attributes are defined as follows:

policyname

The name of the policy.

messagetype

The type of message.

messagecontext

Defines how the message context is set in messages sent by the application:

- The default is `Set By Queue Manager` (the queue manager sets the context).
- If set to `Pass Identity`, the identity of the request message is passed to any output messages.
- If set to `Pass All`, all the context of the request message is passed to any output messages.
- If set to `No Context`, no context is passed.

snd_priority

The priority set in the message, where 0 is the lowest priority and 9 is the highest. When set to `As Transport`, the value from the queue definition is used. You must deselect `As Transport` before you can set a priority value.

snd_persistence

The persistence set in the message, where `Yes` is persistent and `No` is not persistent. When set to `As Transport`, the value from the underlying queue definition is used.

snd_expiry

A period of time (in tenths of a second) after which the message will not be delivered.

snd_retrycount

The number of times a send will be retried if the return code gives a temporary error. Retry is attempted under the following conditions: Queue full, Queue disabled for put, Queue in use.

snd_retry_intrvl

The interval (in milliseconds) between each retry.

snd_newcorrelid

Whether each message is sent with a new correlation ID (except for response messages, where this is set to the Message ID or Correl ID of the request message).

snd_resp_correlid

The ID set in the Correl ID of a response or report message. This is set to either the Message ID or the Correl ID of the request message, as specified.

snd_xcption_action

The action when a message cannot be delivered. When set to `DLQ`, the message is sent to the dead-letter queue. When set to `Discard`, the message is discarded.

snd_report_data

The amount of data included in a report message, where `Report` specifies no data, `With Data` specifies the first 100 bytes, and `With Full Data` specifies all data.

snd_rt_exception

Whether Exception reports are required.

snd_rt_coa

Whether Confirm on Arrival reports are required.

snd_rt_cod

Whether Confirm on Delivery reports are required.

snd_rt_expiry

Whether Expiry reports are required.

reply_q

The name of the reply queue.

reply_qmgr

The name of the reply Queue Manager.

rcv_truncatedmsg

Whether truncated messages are accepted.

rcv_convert

Whether the message is code page converted by the message transport when received.

rcv_poisonmsg

Whether poison message handling is enabled. Sometimes, a badly formatted message arrives on a queue. Such a message might make the receiving application fail and back out the receipt of the message. In this situation, such a message might be received, and then returned to the queue repeatedly.

rcv_openshared

Whether the queue is opened as a shared queue.

rcv_wait_intrvl

A period of time (in milliseconds) that the receive waits for a message to be available.

pub_suppressreg

Whether implicit registration of the publisher is suppressed. (This attribute is ignored for WebSphere® MQ Integrator Version 2.)

pub_anonymous

Whether the publisher registers anonymously.

pub_publocal

Whether the publication is only sent to subscribers that are local to the broker.

pub_direct

Whether the publisher should accept direct requests from subscribers.

pub_correlasid

Whether the Correl ID is used by the broker as part of the publisher's identity.

pub_retain

Whether the publication is retained by the broker.

pub_othersonly

Whether the publication is not sent to the publisher if it has subscribed to the same topic (used for conference-type applications).

sub_anonymous

Whether the subscriber registers anonymously.

sub_sublocal

Whether the subscriber is sent publications that were published with the `Publish Locally` option, at the local broker only.

sub_newpubsonly

Whether the subscriber is not sent existing retained publications when it registers.

sub_pubonreqonly

Whether the subscriber is not sent retained publications, unless it requests them by using Request Update.

sub_correlasid

The broker as part of the subscriber's identity.

sub_informifret

Whether the broker informs the subscriber if a publication is retained.

sub_unsuball

Whether all topics for this subscriber are to be deregistered.

syncpoint

Whether the operation occurred within a syncpoint.

MQCreateVtiRead() function

The MQCreateVtiRead() function creates a table and maps it to a queue managed by WMQ.

Syntax

```
MQCREATEVTIREAD( table_name [ , service_name [ , policy_name [ , maxMessage ] ] ] )
```

table_name

Required parameter. Specifies the name of the table to be created. The queue pointed to by the *service_name* parameter is mapped to this table.

service_name

Optional parameter. Refers to the value in the **servicename** column of the **"informix".mqiservice** table. If *service_name* is not specified, IDS.DEFAULT.SERVICE is used as the service. The maximum size of *service_name* is 48 bytes.

policy_name

Optional parameter. Refers to the value in the **policyname** column of the **"informix".mqipolicy** table. If *policy_name* is not specified, IDS.DEFAULT.POLICY is used as the policy. The maximum size of *policy_name* is 48 bytes.

maxMessage

Optional parameter. Specifies the maximum length of the message to be sent or received. The default value is `4000`; the maximum allowable size is `32628`.

Usage

The MQCreateVtiRead() function creates a table bound to a queue specified by *service_name*, using the quality of service policy defined in *policy_name*. Selecting from the table created by this function returns all the committed messages in the queue, but does not remove the messages from the queue. If no messages are available to be returned, the SELECT statement returns no rows. An insert to the bound table puts a message into the queue.

The table created has the following schema and uses the **"informix".mq** access method:

```
create table table_name (
       msg lvarchar(maxMessage),
       correlid varchar(24),
       topic varchar(40),
       qname varchar(48),
       msgid varchar(12),
       msgformat varchar(8));
             using "informix".mq (SERVICE = service_name,
                                  POLICY = policy_name,
                                  ACCESS = "READ");
```

The mapping for a table bound to a queue requires translation of operation. Actions on specific columns within the table are translated into specific operations within the queue, as outlined here:

- An insert operation inserts the following into the mapped table column:
  - **msg.** The message text that will be inserted onto the queue. If **msg** is NULL, MQ functions send a zero-length message to the queue.
  - **correlid.** The message will be sent with the specified correlation identifier.
- A select operation maps these in the following way to a WMQ queue:
  - **msg.** The message is retrieved from the queue
  - **correlid.** Within the WHERE clause, is the value passed to the queue manager to qualify messages (the correlation identifier). The only operator that should be used when qualifying is equals (=).

The following table describes how the arguments for the MQCreateVtiRead() function are interpreted.

Table 4. MQCreateVtiRead() argument interpretation

Usage	Argument interpretation
MQCreateVtiRead(arg1)	arg1 = *table_name*
MQCreateVtiRead(arg1, arg2)	arg1 = *table_name* arg2 = *service_name*
MQCreateVtiRead(arg1, arg2, arg3)	arg1 = *table_name* arg2 = *service_name* arg3 = *policy_name*
MQCreateVtiRead(arg1, arg2, arg3, arg4)	arg1 = *table_name* arg2 = *service_name* arg3 = *policy_name* arg4 = *maxMessage*

Return codes

't'

The operation was successful.

'f'

The operation was unsuccessful.

Example

Create a table called **VtiReadTest** using the default service name and policy name:

```
begin;
EXECUTE FUNCTION MQCreateVtiRead('VtiReadTest');
commit;
```

Insert a message into the queue:

```
INSERT INTO VtiReadTest(msg) values ('QMessage');
```

Read a message from the queue:

```
select * from VtiReadTest;
```

MQCreateVtiReceive() function

The MQCreateVtiReceive() function creates a table and maps it to a queue managed by WMQ.

Syntax

MQCREATEVTIRECEIVE (*table_name* [,*service_name* [,*policy_name* [,*maxMessage*]]])

table_name

Required parameter. Specifies the name of the table to be created. The queue pointed to by the *service_name* parameter is mapped to this table.

service_name

Optional parameter. Refers to the value in the **servicename** column of the **"informix".mqiservice** table. If *service_name* is not specified, IDS.DEFAULT.SERVICE is used as the service. The maximum size of *service_name* is 48 bytes.

policy_name

Optional parameter. Refers to the value in the **policyname** column of the **"informix".mqipolicy** table. If *policy_name* is not specified, IDS.DEFAULT.POLICY is used as the policy. The maximum size of *policy_name* is 48 bytes.

maxMessage

Optional parameter. Specifies the maximum length of the message to be sent or received. The default value is `4000`; the maximum allowable size is `32628`.

Usage

The MQCreateVtiReceive() function creates a *table_name* bound to a queue specified by *service_name*, using the quality of service policy defined in *policy_name*. Selecting from this table returns all the available messages in the queue and also removes the messages from the queue. If no messages are available to be returned, the no rows are returned. An insert into the bound table puts messages in the queue.

The table created has the following schema and uses the **"informix".mq** access method:

```
create table table_name (
       msg lvarchar(maxMessage),
       correlid varchar(24),
       topic varchar(40),
       qname varchar(48),
       msgid varchar(12),
       msgformat varchar(8));
               using "informix".mq (SERVICE = service_name,
```

```
                                        POLICY = policy_name,
                                        ACCESS = "RECEIVE");
```

The mapping between a table bound to a queue requires translation of operation. Actions on specific columns within the table are translated into specific operations within the queue, as outlined here:

- An insert operation maps the following columns to the MQ manager:
  - **msg.** The text that will be inserted onto the queue. If **msg** is NULL, MQ functions send a zero-length message to the queue.
  - **correlid.** The key recognized by queue manager to get messages from the queue
- A select operation maps the following columns to the MQ manager:
  - **msg.** The message is removed from the queue.
  - **correlid.** Within the WHERE clause, is the value passed to the queue manager to qualify messages (the correlation identifier). The only operator that should be used when qualifying is equals (=).

The following table describes how the arguments for the MQCreateVtiReceive() function are interpreted.

Table 5. MQCreateVtiReceive() argument interpretation

Usage	Argument interpretation
MQCreateVtiReceive(arg1)	arg1 = *table_name*
MQCreateVtiReceive(arg1, arg2)	arg1 = *table_name* arg2 = *service_name*
MQCreateVtiReceive(arg1, arg2, arg3)	arg1 = *table_name* arg2 = *service_name* arg3 = *policy_name*
MQCreateVtiReceive(arg1, arg2, arg3, arg4)	arg1 = *table_name* arg2 = *service_name* arg3 = *policy_name* arg4 = *maxMessage*

Return codes

't'

The operation was successful.

'f'

The operation was unsuccessful.

Example

Create the table **VtiReceiveTest** using the default service name and policy name:

```
begin;
EXECUTE FUNCTION MQCreateVtiRead('VtiReceiveTest');
commit;
```

Insert a message to the queue:

```
INSERT INTO VtiReceiveTest(msg) values ('QMessage');
```

Read a message from the queue:

```
select * from VtiReceiveTest;
```

Attempting to read the queue a second time results in returning no rows because the table was created using the MQCreateVtiReceive() function, which removes entries as they are read.

MQCreateVtiWrite() function

The MQCreateVtiWrite() function creates a write-only VTI table and maps it to a queue that manages.

Syntax

MQCreateVtiWrite (*table_name* [, *service_name* [, *policy_name* [, *maxMessage*]]])

table_name

Required parameter. Specifies the name of the table to be created. The queue pointed to by the *service_name* parameter is mapped to this table.

service_name

Optional parameter. Refers to the value in the **servicename** column of the **"informix".mqiservice** table. If *service_name* is not specified, IDS.DEFAULT.SERVICE is used as the service. The maximum size of *service_name* is 48 bytes.

policy_name

Optional parameter. Refers to the value in the **policyname** column of the **"informix".mqipolicy** table. If *policy_name* is not specified, IDS.DEFAULT.POLICY is used as the policy. The maximum size of *policy_name* is 48 bytes.

maxMessage

Optional parameter. Specifies the maximum length of the message to be sent or received. The default value is `4000`; the maximum allowable size is `32628`. If the value is `-1`, the message is a CLOB data type. If the value is `-2`, the message is a BLOB data type.

Usage

You can perform only an insert operation on this table. You cannot perform a select operation on this table.

The following table describes how the arguments for the MQCreateVtiWrite() function are interpreted.

Table 6. MQCreateVtiWrite() argument interpretation

Usage	Argument interpretation
MQCreateVtiWrite(arg1)	arg1 = *table_name*
MQCreateVtiWrite(arg1, arg2)	arg1 = *table_name* arg2 = *service_name*
MQCreateVtiWrite(arg1, arg2, arg3)	arg1 = *table_name* arg2 = *service_name* arg3 = *policy_name*
MQCreateVtiWrite(arg1, arg2, arg3, arg4)	arg1 = *table_name* arg2 = *service_name* arg3 = *policy_name* arg4 = *maxMessage*

Example

The following example creates a table named `qm0vti` for service `lser.qm1`.

```
execute function MQCreateVtiRead("qm0vti", "lser.qm1");
```

MQHasMessage() function

The MQHasMessage() function checks if a message is available from the WMQ.

Syntax

```
MQHasMessage( [ service_name [ ,policy_name [ ,correl_id ] ] ] )
```

service_name

Optional parameter. Refers to the value in the **servicename** column of the **"informix".mqiservice** table. If *service_name* is not specified, IDS.DEFAULT.SERVICE is used as the service. The maximum size of *service_name* is 48 bytes.

policy_name

Optional parameter. Refers to the value in the **policyname** column of the **"informix".mqipolicy** table. If *policy_name* is not specified, IDS.DEFAULT.POLICY is used as the policy. The maximum size of *policy_name* is 48 bytes.

correl_id

Optional parameter. A string containing a correlation identifier to be associated with this message. The *correl_id* is often specified in request and reply scenarios to associate requests with replies. The maximum size of *correl_id* is 24 bytes. If not specified, no correlation ID is added to the message.

Usage

You can simulate event processing by using this function and other MQ functions to write custom procedures and run them inside the Scheduler at specified intervals.

The following table describes how the arguments for the MQHasMessage() function are interpreted.

Table 7. MQHasMessage() argument interpretation

Usage	Argument interpretation
MQHasMessage()	No arguments
MQHasMessage(arg1)	arg1 = *service_name*
MQHasMessage(arg1, arg2)	arg1 = *service_name* arg2 = *policy_name*
MQHasMessage(arg1, arg2, arg3)	arg1 = *service_name* arg2 = *policy_name* arg3 = *correl_id*

Return codes

1

One message or more than one message is present.

0

No Messages are available.

Error

The operation was unsuccessful.

Example

This following example reads the message with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *correl_id*: "TESTS"

```
begin;
EXECUTE FUNCTION MQHasMessage('MYSERVICE','MYPOLICY', 'TESTS');
commit;
```

MQInquire() function

The MQInquire() function, which is the same as the MQINQ() function of , queries attributes of the queue. The MQInquire() is the interface between your SQL and .

Syntax

```
MQInquire( [ service_name, ] selector )
```

service_name

Optional parameter. Refers to the value in the **servicename** column of the **"informix".mqiservice** table. If *service_name* is not specified, IDS.DEFAULT.SERVICE is used as the service. The maximum size of *service_name* is 48 bytes.

selector

An integer or character attribute selectors number or string, such as MQCA_* or MQIA_* values that exist in WMQ product documentation or header files. Examples of string values are MQIA_Q_TYPE or MQIA_CURRENT_Q_DEPTH.

Usage

The following table describes how the arguments for the MQInquire() function are interpreted.

Table 8. MQInquire() argument interpretation

Usage	Argument interpretation
MQInquire(arg1)	arg1 = *selector (number or string)*
MQInquire(arg1, arg2)	arg1 = *service_name* arg2 = *(number or string)*

You can use the MQINQ() selectors of the .

Return codes

A string of LVARCHAR type

The operation was successful.

NULL

No Messages are available.

Error

The operation was unsuccessful.

Examples

The following example shows an integer selector for a queue type:

```
execute function MQInquire('IDS.DEFAULT.SERVICE',20);  -- Queue Type
```

The following example shows a character attribute selector for a queue type:

```
execute function MQInquire('MQIA_Q_TYPE');
```

The following example shows a string selector for queue depth:

```
execute function MQInquire('IDS.DEFAULT.SERVICE',3);
```

The following example shows a character attribute selector for queue depth:

```
execute function MQInquire('IDS.DEFAULT.SERVICE','
 MQIA_CURRENT_Q_DEPTH');
```

MQPublish() function

The MQPublish() function publishes a message on one or more topics to a queue managed by WMQ.

Syntax

MQPUBLISH([*publisher_name*, [*policy_name*,]] *msg_data* [, *topic*] [, *correl_id*]) (explicit id)

publisher_name

Optional parameter. Refers to the value in the **pubsubname** column of the **"informix".mqipubsub** table. If *publisher_name* is not specified, IDS.DEFAULT.PUBLISHER is used as the publisher. The maximum length of *publisher_name* is `48` bytes.

policy_name

Optional parameter. Refers to the value in the **policyname** column of the **"informix".mqipolicy** table. If *policy_name* is not specified, IDS.DEFAULT.PUB.SUB.POLICY is used as the policy. The maximum size of *policy_name* is 48 bytes.

msg_data

Required parameter. A string containing the data to be sent by WMQ. The maximum size of the string is defined by the LVARCHAR data type. If *msg_data* is NULL, it sends a zero-length message to the queue.

topic

Optional parameter. A string containing the topic for the message publication. The maximum size of a topic is 40 bytes. Multiple topics can be specified in one string (up to 40 characters long). Each topic must be separated by a colon. For example, "t1:t2:the third topic" indicates that the message is associated with all three topics: t1, t2, and the third topic. If no topic is specified, none are associated with the message.

correl_id

Optional parameter. A string containing a correlation identifier to be associated with this message. The *correl_id* is often specified in request and reply scenarios to associate requests with replies. The maximum size of *correl_id* is 24 bytes. If not specified, no correlation ID is added to the message.

Usage

The MQPublish() function publishes data to WMQ. It requires the installation of the WMQ Publish/Subscribe component of WMQ, and that the Message Broker is running.

The MQPublish() function publishes the data contained in *msg_data* to the WMQ publisher specified in *publisher_name*, using the quality of service policy defined by *policy_name*.

The following table describes how the arguments for the MQPublish() function are interpreted.

Table 9. MQPublish() argument interpretation

Usage	Argument interpretation
MQPublish(arg1)	arg1 = *msg_data*
MQPublish(arg1, arg2)	arg1 = *msg_data* arg2 = *topic*
MQPublish(arg1, arg2, arg3)	arg1 = *publisher_name* arg2 = *msg_data* arg3 = *topic*
MQPublish(arg1, arg2, arg3, arg4)	arg1 = *publisher_name* arg2 = *policy_name* arg3 = *msg_data* arg4 = *topic*
MQPublish(arg1, arg2, arg3, arg4, arg5)	arg1 = *publisher_name* arg2 = *policy_name* arg3 = *msg_data* arg4 = *topic* arg5 = *correl_id*

Return codes

1

The operation was successful.

Error

The operation was unsuccessful.

Examples

Example 1

```
begin;
EXECUTE FUNCTION MQPublish('Testing 123');
commit:
```

This example publishes the message with the following parameters:

- *publisher_name*: default publisher
- *policy_name*: default policy
- *msg_data*: "Testing 123"
- *topic*: None
- *correl_id*: None

Example 2

```
begin;
EXECUTE FUNCTION MQPublish
('MYPUBLISHER','Testing 345','TESTTOPIC');
commit;
```

This example publishes the message with the following parameters:

- *publisher_name*: "MYPUBLISHER"
- *policy_name*: default policy
- *msg_data*: "Testing 345"
- *topic*: "TESTTOPIC"
- *correl_id*: None

Example 3

```
begin;
EXECUTE FUNCTION MQPublish('MYPUBLISHER',
'MYPOLICY','Testing 678','TESTTOPIC','TEST1');
commit;
```

This example publishes the message with the following parameters:

- *publisher_name*: "MYPUBLISHER"
- *policy_name*: "MYPOLICY"
- *msg_data*: "Testing 678"
- *topic*: "TESTTOPIC"
- *correl_id*: "TEST1"

Example 4

```
begin;
EXECUTE FUNCTION MQPublish('Testing 901','TESTS');
commit;
```

This example publishes the message with the following parameters:

- *publisher_name*: default publisher
- *policy_name*: default policy
- *msg_data*: "Testing 901"
- *topic*: "TESTS"
- *correl_id*: None

Example 5

```
begin;
EXECUTE FUNCTION MQPublish('SEND.MESSAGE',
'emergency', 'CODE BLUE', 'expedite');
commit;
```

This example publishes the message with the following parameters:

- *publisher_name*: "SEND.MESSAGE"
- *policy_name*: "emergency"
- *msg_data*: "CODE BLUE"
- *topic*: "expedite"
- *correl_id*: None

Example 6

The following table contains sample rows and columns in the **"informix".mqipubsub** table.

Sample row	pubsubname column	receiver column	pubsubtype column
Sample row 1	'IDS.DEFAULT. PUBLISHER'	' '	'Publisher'
Sample row 2	'IDS.DEFAULT. SUBSCRIBER'	'IDS.DEFAULT. SUBSCRIBER.RECEIVER'	'Subscriber'

```
begin;
 EXECUTE FUNCTION
  MQSubscribe('IDS.DEFAULT.SUBSCRIBER',
              'IDS.DEFAULT.PUB.SUB.POLICY', 'Weather');
commit;
```

This statement demonstrates a subscriber registering an interest in messages containing the topic *"Weather,"* with the following parameters:

- *subscriber_name*: "IDS.DEFAULT.SUBSCRIBER"
- *policy_name*: "IDS.DEFAULT.PUB.SUB.POLICY"
- *topic*: "Weather"

```
begin;
   EXECUTE FUNCTION MQPublish
    ('IDS.DEFAULT.PUBLISHER',
     'IDS.DEFAULT.PUB.SUB.POLICY', 'Rain', 'Weather');
commit;
```

This statement publishes the message with the following parameters:

- *publisher_name*: "IDS.DEFAULT.PUBLISHER"
- *policy_name*: "IDS.DEFAULT.PUB.SUB.POLICY"
- *msg_data*: "Rain"
- *topic*: "Weather"
- *correl_id*: None

```
begin;
   EXECUTE FUNCTION MQReceive
      ('IDS.DEFAULT.SUBSCRIBER.RECEIVER',
       'IDS.DEFAULT.PUB.SUB.POLICY');
commit;
```

This statement receives the message with the following parameters (it returns "Rain"):

- *service_name*: "IDS.DEFAULT.SUBSCRIBER.RECEIVER"
- *policy_name*: "IDS.DEFAULT.PUB.SUB.POLICY"

MQPublishClob() function

The MQPublishClob() function publishes CLOB data on one or more topics to a queue managed by WMQ.

Syntax

MQPUBLISHCLOB([*publisher_name*, [*policy_name*,]] *clob_data* [, *topic*] [, *correl_id*]) (explicit id)

publisher_name

Optional parameter. Refers to the value in the **pubsubname** column of the **"informix".mqipubsub** table. If *publisher_name* is not specified, IDS.DEFAULT.PUBLISHER is used as the publisher. The maximum length of *publisher_name* is `48` bytes.

policy_name

Optional parameter. Refers to the value in the **policyname** column of the **"informix".mqipolicy** table. If *policy_name* is not specified, IDS.DEFAULT.PUB.SUB.POLICY is used as the policy. The maximum size of *policy_name* is 48 bytes.

clob_data

Required parameter. The CLOB data to be sent to WMQ. Even though the CLOB data size can be up to 4 TB, the maximum size of the message is limited by what Websphere MQ supports. If *clob_data* is NULL, it sends a zero-length message to the queue.

topic

Optional parameter. A string containing the topic for the message publication. The maximum size of a topic is 40 bytes. Multiple topics can be specified in one string (up to 40 characters long). Each topic must be separated by a colon. For example, "t1:t2:the third topic" indicates that the message is associated with all three topics: t1, t2, and the third topic. If no topic is specified, none are associated with the message.

correl_id

Optional parameter. A string containing a correlation identifier to be associated with this message. The *correl_id* is often specified in request and reply scenarios to associate requests with replies. The maximum size of *correl_id* is 24 bytes. If not specified, no correlation ID is added to the message.

Usage

The MQPublishClob() function publishes data to WMQ. It requires the installation of the WMQ Publish/Subscribe component of WMQ, and that the Message Broker is running.

The MQPublishClob() function publishes the data contained in *clob_data* to the WMQ publisher specified in *publisher_name*, using the quality of service policy defined by *policy_name*.

The following table describes how the arguments for the MQPublishClob() function are interpreted.

Table 10. MQPublishClob() argument interpretation

Usage	Argument interpretation
MQPublishClob(arg1)	arg1 = *clob_data*
MQPublishClob(arg1, arg2)	arg1 = *clob_data* arg2 = *topic*
MQPublishClob(arg1, arg2, arg3)	arg1 = *publisher_name* arg2 = *clob_data* arg3 = *topic*
MQPublishClob(arg1, arg2, arg3, arg4)	arg1 = *publisher_name*

Table 10. MQPublishClob() argument interpretation (continued)

Usage	Argument interpretation
	arg2 = *policy_name* arg3 = *clob_data* arg4 = *topic*
MQPublishClob(arg1, arg2, arg3, arg4, arg5)	arg1 = *publisher_name* arg2 = *policy_name* arg3 = *clob_data* arg4 = *topic* arg5 = *correl_id*

Return codes

1

The operation was successful.

Error

The operation was unsuccessful.

Examples

Example 1

```
begin;
EXECUTE FUNCTION MQPublishClob(filetoclob("/work/mydata",
"client");
commit:
```

This example publishes the message with the following parameters:

- *publisher_name*: default publisher
- *policy_name*: default policy
- *clob_data*: filetoclob("/work/mydata", "client")
- *topic*: None
- *correl_id*: None

Example 2

```
begin;
EXECUTE FUNCTION MQPublishClob('MYPUBLISHER',
filetoclob("/work/mydata", "client"),'TESTTOPIC');
commit;
```

This example publishes the message with the following parameters:

- *publisher_name*: "MYPUBLISHER"
- *policy_name*: default policy
- *clob_data*: filetoclob("/work/mydata", "client")
- *topic*: "TESTTOPIC"
- *correl_id*: None

Example 3

```
begin;
EXECUTE FUNCTION MQPublishClob('MYPUBLISHER',
'MYPOLICY',filetoclob("/work/mydata",
"client"),'TESTTOPIC','TEST1');commit;
```

This example publishes the message with the following parameters:

- *publisher_name*: "MYPUBLISHER"
- *policy_name*: "MYPOLICY"
- *clob_data*: filetoclob("/work/mydata", "client")
- *topic*: "TESTTOPIC"
- *correl_id*: "TEST1"

Example 4

```
begin;
EXECUTE FUNCTION MQPublishClob
(filetoclob("/work/mydata", "client"),'TESTS');
commit;
```

This example publishes the message with the following parameters:

- *publisher_name*: default publisher
- *policy_name*: default policy
- *clob_data*: filetoclob("/work/mydata", "client")
- *topic*: "TESTS"
- *correl_id*: None

Example 5

```
begin;
EXECUTE FUNCTION MQPublishClob('SEND.MESSAGE',
 'emergency', filetoclob("/work/mydata", "client")
 'expedite');commit;
```

This example publishes the message with the following parameters:

- *publisher_name*: "SEND.MESSAGE"
- *policy_name*: "emergency"
- *clob_data*: filetoclob("/work/mydata", "client")
- *topic*: "expedite"
- *correl_id*: None

Example 6

The following table contains sample rows and columns in the **"informix".mqipubsub** table.

Sample row	pubsubname column	receiver column	pubsubtype column
Sample row 1	'IDS.DEFAULT. PUBLISHER'	''	'Publisher'
Sample row 2	'IDS.DEFAULT. SUBSCRIBER'	'IDS.DEFAULT. SUBSCRIBER.RECEIVER'	'Subscriber'

```
begin;
 EXECUTE FUNCTION
    MQSubscribe('IDS.DEFAULT.SUBSCRIBER',
                'IDS.DEFAULT.PUB.SUB.POLICY', 'Weather');
commit;
```

This statement demonstrates a subscriber registering an interest in messages containing the topic "Weather," with the following parameters:

- *subscriber_name*: "IDS.DEFAULT.SUBSCRIBER"
- *policy_name*: "IDS.DEFAULT.PUB.SUB.POLICY"
- *topic*: "Weather"

```
begin;
   EXECUTE FUNCTION MQPublishClob('IDS.DEFAULT.PUBLISHER',
                            'IDS.DEFAULT.PUB.SUB.POLICY',
 filetoclob("/work/mydata",
 "client"), 'Weather');commit;
```

This statement publishes the message with the following parameters:

- *publisher_name*: "IDS.DEFAULT.PUBLISHER"
- *policy_name*: "IDS.DEFAULT.PUB.SUB.POLICY"
- *clob_data*: filetoclob("/work/mydata", "client")
- *topic*: "Weather"
- *correl_id*: None

```
begin;
   EXECUTE FUNCTION MQReceiveClob('IDS.DEFAULT.SUBSCRIBER.RECEIVER',
                              'IDS.DEFAULT.PUB.SUB.POLICY');
commit;
```

This statement receives the message with the following parameters:

- *service_name*: "IDS.DEFAULT.SUBSCRIBER.RECEIVER"
- *policy_name*: "IDS.DEFAULT.PUB.SUB.POLICY"

MQRead() function

The MQRead() function returns a message from WMQ without removing the message from the queue.

Syntax

MQREAD([*service_name* [, *policy_name* [, *correl_id*]]])

service_name

Optional parameter. Refers to the value in the **servicename** column of the **"informix".mqiservice** table. If *service_name* is not specified, IDS.DEFAULT.SERVICE is used as the service. The maximum size of *service_name* is 48 bytes.

policy_name

Optional parameter. Refers to the value in the **policyname** column of the **"informix".mqipolicy** table. If *policy_name* is not specified, IDS.DEFAULT.POLICY is used as the policy. The maximum size of *policy_name* is 48 bytes.

correl_id

Optional parameter. A string containing a correlation identifier to be associated with this message. The *correl_id* is often specified in request and reply scenarios to associate requests with replies. The maximum size of *correl_id* is 24 bytes. If not specified, no correlation ID is added to the message.

Usage

The MQRead() function returns a message from the WMQ queue specified by *service_name*, using the quality of service policy defined in *policy_name*. This function does not remove the message from the queue associated with *service_name*. If *correl_id* is specified, then the first message with a matching correlation ID is returned. If *correl_id* is not specified, then the message at the head of the queue is returned. The result of the function is a string of type LVARCHAR. If no messages are returned, this function returns NULL. This function only reads committed messages.

The following table describes how the arguments for the MQRead() function are interpreted.

Table 11. MQRead() argument interpretation

Usage	Argument interpretation
MQRead()	No arguments

Table 11. MQRead() argument interpretation (continued)

Usage	Argument interpretation
MQRead(arg1)	arg1 = *service_name*
MQRead(arg1, arg2)	arg1 = *service_name* arg2 = *policy_name*
MQRead(arg1, arg2, arg3)	arg1 = *service_name* arg2 = *policy_name* arg3 = *correl_id*

Return codes

A string of type LVARCHAR

The operation was successful.

NULL

No Messages are available.

Error

The operation was unsuccessful.

Examples

Example 1

```
begin;
EXECUTE FUNCTION MQRead();
commit;
```

Alternatively, the following syntax can be used:

```
insert into my_order_table VALUES(MQRead());
```

This example reads the message at the head of the queue with the following parameters:

- *service_name*: default service name
- *policy_name*: default policy name
- *correl_id*: None

Example 2

```
begin;
EXECUTE FUNCTION MQRead('MYSERVICE');
rollback;
```

Alternatively, the following syntax can be used:

```
insert into my_order_table VALUES(MQRead('MYSERVICE'));
```

This example reads the message at the head of the queue with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: default policy name
- *correl_id*: None

Example 3

```
begin;
EXECUTE FUNCTION MQRead('MYSERVICE','MYPOLICY');
commit;
```

Alternatively, the following syntax can be used:

```
insert into my_order_table VALUES(MQRead('MYSERVICE', 'MYPOLICY'));
```

This example reads the message at the head of the queue with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *correl_id*: None

Example 4

```
begin;
EXECUTE FUNCTION MQRead('MYSERVICE','MYPOLICY', 'TESTS');
commit;
```

Alternatively, the following syntax can be used:

```
insert into my_order_table VALUES(MQRead('MYSERVICE', 'MYPOLICY', 'TESTS'));
```

This example reads the message at the head of the queue with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *correl_id*: "TESTS"

MQReadClob() function

The MQReadClob() function returns a message as a CLOB from WMQ without removing the message from the queue.

Syntax

MQREADCLOB([*service_name* [,*policy_name* [,*correl_id*]]])

service_name

Optional parameter. Refers to the value in the **servicename** column of the **"informix".mqiservice** table. If *service_name* is not specified, IDS.DEFAULT.SERVICE is used as the service. The maximum size of *service_name* is 48 bytes.

policy_name

Optional parameter. Refers to the value in the **policyname** column of the **"informix".mqipolicy** table. If *policy_name* is not specified, IDS.DEFAULT.POLICY is used as the policy. The maximum size of *policy_name* is 48 bytes.

correl_id

Optional parameter. A string containing a correlation identifier to be associated with this message. The *correl_id* is often specified in request and reply scenarios to associate requests with replies. The maximum size of *correl_id* is 24 bytes. If not specified, no correlation ID is added to the message.

Usage

The MQReadClob() function returns a message as a CLOB from the WMQ location specified by *service_name*, using the quality-of-service policy defined in *policy_name*. This function does not remove the message from the queue associated with *service_name*. If *correl_id* is specified, then the first message with a matching correlation ID is returned. If *correl_id* is not specified, then the message at the head of the queue is returned. The result of this function is a CLOB type. If no messages are available to be returned, this function returns NULL. This function only reads committed messages.

The following table describes how the arguments for the MQReadClob() function are interpreted.

Table 12. MQReadClob() argument interpretation

Usage	Argument interpretation
MQReadClob()	No arguments
MQReadClob(arg1)	arg1 = *service_name*
MQReadClob(arg1, arg2)	arg1 = *service_name* arg2 = *policy_name*
MQReadClob(arg1, arg2, arg3)	arg1 = *service_name* arg2 = *policy_name* arg3 = *correl_id*

Return codes

The contents of the message as a CLOB

The operation was successful. If no messages are available, the result is NULL.

Error

The operation was unsuccessful.

Example

Example 1

```
begin;
EXECUTE FUNCTION MQReadClob();
commit;
```

Alternatively, the following syntax can be used:

```
insert into my_order_table(clob_col) VALUES(MQReadClob());
```

This example reads the content of the message as a CLOB at the head of the queue into the CLOB with the following parameters:

- *service_name*: default service name
- *policy_name*: default policy name
- *correl_id*: None

Example 2

```
begin;
EXECUTE FUNCTION MQReadClob('MYSERVICE');
rollback;
```

Alternatively, the following syntax can be used:

```
insert into my_order_table(clob_col)
 VALUES(MQReadClob('MYSERVICE'));
```

This example reads the content of the message as a CLOB at the head of the queue into the CLOB with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: default policy name
- *correl_id*: None

Example 3

```
begin;
EXECUTE FUNCTION MQReadClob('MYSERVICE','MYPOLICY');
commit;
```

Alternatively, the following syntax can be used:

```
insert into my_order_table(clob_col)
 VALUES(MQReadClob('MYSERVICE', 'MYPOLICY'));
```

This example reads the content of the message as a CLOB at the head of the queue into the CLOB with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *correl_id*: None

Example 4

```
begin;
EXECUTE FUNCTION MQReadClob('MYSERVICE','MYPOLICY', 'TESTS');
commit;
```

Alternatively, the following syntax can be used:

```
insert into my_order_table(clob_col)
 VALUES(MQReadClob('MYSERVICE', 'MYPOLICY', 'TESTS'));
```

This example reads the content of the message as a CLOB at the head of the queue into the CLOB with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *correl_id*: "TESTS"

MQReceive() function

The MQReceive() function returns a message from the WMQ queue and removes the message from the queue.

Syntax

```
MQRECEIVE( [ service_name [ ,policy_name [ ,correl_id ] ] ] )
```

service_name

Optional parameter. Refers to the value in the **servicename** column of the **"informix".mqiservice** table. If *service_name* is not specified, IDS.DEFAULT.SERVICE is used as the service. The maximum size of *service_name* is 48 bytes.

policy_name

Optional parameter. Refers to the value in the **policyname** column of the **"informix".mqipolicy** table. If *policy_name* is not specified, IDS.DEFAULT.POLICY is used as the policy. The maximum size of *policy_name* is 48 bytes.

correl_id

Optional parameter. A string containing a correlation identifier to be associated with this message. The *correl_id* is often specified in request and reply scenarios to associate requests with replies. The maximum size of *correl_id* is 24 bytes. If not specified, no correlation ID is added to the message.

Usage

The MQReceive() function returns a message from the WMQ location specified by *service_name*, using the quality of service policy *policy_name*. This function removes the message from the queue associated with *service_name*. If *correl_id*

is specified, then the first message with a matching correlation identifier is returned. If *correl_id* is not specified, then the message at the head of the queue is returned. The result of the function is a string LVARCHAR type. If no messages are available to be returned, the function returns NULL.

The following table describes how the arguments for the MQReceive() function are interpreted.

Table 13. MQReceive() argument interpretation

Usage	Argument interpretation
MQReceive()	No arguments
MQReceive(arg1)	arg1 = *service_name*
MQReceive(arg1, arg2)	arg1 = *service_name* arg2 = *policy_name*
MQReceive(arg1, arg2, arg3)	arg1 = *service_name* arg2 = *policy_name* arg3 = *correl_id*

Return codes

A string of LVARCHAR type

The operation was successful.

NULL

No messages are available.

Error

The operation was unsuccessful.

Examples

Example 1

```
begin;
EXECUTE FUNCTION MQReceive();
commit;
```

Alternatively, the following syntax can be used:

```
insert into my_order_table VALUES(MQReceive());
```

This example receives the message at the head of the queue with the following parameters:

- *service_name*: default service name
- *policy_name*: default policy name
- *correl_id*: none

Example 2

```
begin;
EXECUTE FUNCTION MQReceive('MYSERVICE');
rollback;
```

Alternatively, the following syntax can be used:

```
insert into my_order_table VALUES(MQReceive('MYSERVICE'));
```

This example receives the message at the head of the queue with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: default policy name
- *correl_id*: none

Example 3

```
begin;
EXECUTE FUNCTION MQReceive('MYSERVICE','MYPOLICY');
commit;
```

Alternatively, the following syntax can be used:

```
insert into my_order_table VALUES(MQReceive('MYSERVICE', 'MYPOLICY'));
```

This example receives the message at the head of the queue with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *correl_id*: none

Example 4

```
begin;
EXECUTE FUNCTION MQReceive('MYSERVICE','MYPOLICY','1234');
commit;
```

Alternatively, the following syntax can be used:

```
insert into my_order_table VALUES(MQReceive('MYSERVICE', 'MYPOLICY', '1234'));
```

This example receives the message at the head of the queue with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *correl_id*: "1234"

MQReceiveClob() function

The MQReceiveClob() function retrieves a message as a CLOB from the WMQ queue and removes the message from the queue.

Syntax

```
MQRECEIVECLOB( [ service_name [ ,policy_name [ ,correl_id ] ] ] )
```

service_name

Optional parameter. Refers to the value in the **servicename** column of the **"informix".mqiservice** table. If *service_name* is not specified, IDS.DEFAULT.SERVICE is used as the service. The maximum size of *service_name* is 48 bytes.

policy_name

Optional parameter. Refers to the value in the **policyname** column of the **"informix".mqipolicy** table. If *policy_name* is not specified, IDS.DEFAULT.POLICY is used as the policy. The maximum size of *policy_name* is 48 bytes.

correl_id

Optional parameter. A string containing a correlation identifier to be associated with this message. The *correl_id* is often specified in request and reply scenarios to associate requests with replies. The maximum size of *correl_id* is 24 bytes. If not specified, no correlation ID is added to the message.

Usage

The MQReceiveClob() function returns a message as a CLOB from the WMQ location specified by *service_name*, using the quality-of-service policy *policy_name*. This function removes the message from the queue associated with *service_name*. If *correl_id* is specified, then the first message with a matching correlation identifier is returned. If *correl_id* is not specified, then the message at the head of the queue is returned. The result of the function is a CLOB. If messages are not available to be returned, the function returns NULL.

The following table describes how the arguments for the MQReceiveClob() function are interpreted.

Table 14. MQReceiveClob() argument interpretation

Usage	Argument interpretation
MQReceiveClob()	No arguments
MQReceiveClob(arg1)	arg1 = *service_name*
MQReceiveClob(arg1, arg2)	arg1 = *service_name* arg2 = *policy_name*
MQReceiveClob(arg1, arg2, arg3)	arg1 = *service_name* arg2 = *policy_name*

Table 14. MQReceiveClob() argument interpretation (continued)

Usage	Argument interpretation
	arg3 = *correl_id*

Return codes

The contents of the message as a CLOB

The operation was successful. If no messages are available, the result is NULL.

Error

The operation was unsuccessful.

Examples

Example 1

```
begin;
EXECUTE FUNCTION MQReceiveClob();
commit;
```

Alternatively, the following syntax can be used:

```
insert into my_order_table(clob_col) VALUES(MQReceiveClob());
```

This example receives the content of the message as a CLOB at the head of the queue into the CLOB with the following parameters:

- *service_name*: default service name
- *policy_name*: default policy name
- *correl_id*: none

Example 2

```
begin;
EXECUTE FUNCTION MQReceiveClob('MYSERVICE');
rollback;
```

Alternatively, the following syntax can be used:

```
insert into my_order_table(clob_col)
 VALUES(MQReceiveClob('MYSERVICE'));
```

This example receives the content of the message as a CLOB at the head of the queue into the CLOB with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: default policy name
- *correl_id*: none

Example 3

```
begin;
EXECUTE FUNCTION MQReceiveClob('MYSERVICE', 'MYPOLICY');
commit;
```

Alternatively, the following syntax can be used:

```
insert into my_order_table(clob_col)
 VALUES(MQReceiveClob('MYSERVICE', 'MYPOLICY'));
```

This example receives the content of the message as a CLOB at the head of the queue into the CLOB with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *correl_id*: none

Example 4

```
begin;
EXECUTE FUNCTION MQReceiveClob('MYSERVICE', 'MYPOLICY', 'TESTS');
commit;
```

Alternatively, the following syntax can be used:

```
insert into my_order_table(clob_col)
 VALUES(MQReceiveClob('MYSERVICE', 'MYPOLICY', 'TESTS'));
```

This example receives the content of the message as a CLOB at the head of the queue into the CLOB with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *correl_id*: "TESTS"

MQSend() function

The MQSend() function puts the message into the WMQ queue.

Syntax

MQSEND([*service_name*, [*policy_name*,]] *msg_data* [, *correl_id*]) (explicit id)

service_name

Optional parameter. Refers to the value in the **servicename** column of the **"informix".mqiservice** table. If *service_name* is not specified, IDS.DEFAULT.SERVICE is used as the service. The maximum size of *service_name* is 48 bytes.

policy_name

Optional parameter. Refers to the value in the **policyname** column of the **"informix".mqipolicy** table. If *policy_name* is not specified, IDS.DEFAULT.POLICY is used as the policy. The maximum size of *policy_name* is 48 bytes.

msg_data

Required parameter. A string containing the data to be sent by WMQ. The maximum size of the string is defined by the LVARCHAR data type. If *msg_data* is NULL, it sends a zero-length message to the queue.

correl_id

Optional parameter. A string containing a correlation identifier to be associated with this message. The *correl_id* is often specified in request and reply scenarios to associate requests with replies. The maximum size of *correl_id* is 24 bytes. If not specified, no correlation ID is added to the message.

Usage

The MQSend() function puts the data contained in *msg_data* into the WMQ location specified by *service_name*, using the quality of policy name defined by *policy_name*. If *correl_id* is specified, then the message is sent with a correlation identifier. If *correl_id* is not specified, then no correlation ID is sent with the message.

The following table describes how the arguments for the MQSend() function are interpreted.

Table 15. MQSend() argument interpretation

Usage	Argument interpretation
MQSend(arg1)	arg1 = *msg_data*
MQSend(arg1, arg2)	arg1 = *service_name* arg2 = *msg_data*
MQSend(arg1, arg2, arg3)	arg1 = *service_name* arg2 = *policy_name* arg3 = *msg_data*
MQSend(arg1, arg2, arg3, arg4)	arg1 = *service_name* arg2 = *policy_name* arg3 = *msg_data* arg4 = *correl_id*

Return codes

1

The operation was successful.

0 or Error

The operation was unsuccessful.

Examples

Example 1

```
EXECUTE FUNCTION MQSend('Testing 123')
```

This example sends the message to the WMQ with the following parameters:

- *service_name*: default service name
- *policy_name*: default policy
- *msg_data*: "Testing 123"
- *correl_id*: none

Example 2

```
begin;
EXECUTE FUNCTION MQSend('MYSERVICE','Testing 901');
commit;
```

This example sends the message to the WMQ with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: default policy
- *msg_data*: "Testing 901"
- *correl_id*: none

Example 3

```
begin;
EXECUTE FUNCTION MQSend('MYSERVICE','MYPOLICY','Testing 345');
commit;
```

This example sends the message to the WMQ with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *msg_data*: "Testing 345"
- *correl_id*: none

Example 4

```
begin;
EXECUTE FUNCTION MQSend('MYSERVICE','MYPOLICY','Testing 678','TEST3');
commit;
```

This example sends the message to the WMQ with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *msg_data*: "Testing 678"
- *correl_id*: "TEST3"

MQSendClob() function

The MQSendClob() function puts the CLOB data into the WMQ queue.

Syntax

MQSENDCLOB([*service_name*, [*policy_name*,]] *clob_data* [, *correl_id*]) (explicit id)

service_name

Optional parameter. Refers to the value in the **servicename** column of the **"informix".mqiservice** table. If *service_name* is not specified, IDS.DEFAULT.SERVICE is used as the service. The maximum size of *service_name* is 48 bytes.

policy_name

Optional parameter. Refers to the value in the **policyname** column of the **"informix".mqipolicy** table. If *policy_name* is not specified, IDS.DEFAULT.POLICY is used as the policy. The maximum size of *policy_name* is 48 bytes.

clob_data

Required parameter. The CLOB data to be sent to WMQ. Even though the CLOB data size can be up to 4 TB, the maximum size of the message is limited by what Websphere MQ supports. If *clob_data* is NULL, it sends a zero-length message to the queue.

correl_id

Optional parameter. A string containing a correlation identifier to be associated with this message. The *correl_id* is often specified in request and reply scenarios to associate requests with replies. The maximum size of *correl_id* is 24 bytes. If not specified, no correlation ID is added to the message.

Usage

The MQSendClob() function puts the data contained in *clob_data* to the WMQ queue specified by *service_name*, using the quality of service policy defined by *policy_name*. If *correl_id* is specified, then the message is sent with a correlation identifier. If *correl_id* is not specified, then no correlation ID is sent with the message.

The following table describes how the arguments for the MQSendClob() function are interpreted.

Table 16. MQSendClob() argument interpretation

Usage	Argument interpretation
MQSendClob(arg1)	arg1 = *clob_data*
MQSendClob(arg1, arg2)	arg1 = *service_name* arg2 = *clob_data*
MQSendClob(arg1, arg2, arg3)	arg1 = *service_name* arg2 = *policy_name* arg3 = *clob_data*
MQSendClob(arg1, arg2, arg3, arg4)	arg1 = *service_name* arg2 = *policy_name* arg3 = *clob_data* arg4 = *correl_id*

Return codes

1

The operation was successful.

0 or Error

The operation was unsuccessful.

Examples

Example 1

```
begin;
EXECUTE FUNCTION MQSendClob(filetoclob("/work/mydata", "client"));
commit;
```

This example sends a CLOB to the WMQ with the following parameters:

- *service_name*: default service name
- *policy_name*: default policy
- *clob_data*: filetoclob("/work/mydata", "client")
- *correl_id*: none

Example 2

```
begin;
EXECUTE FUNCTION MQSendClob('MYSERVICE', filetoclob("/work/mydata", "client"));
commit;
```

This example sends a CLOB to the WMQ with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: default policy
- *msg_data*: filetoclob("/work/mydata", "client")
- *correl_id*: none

Example 3

```
begin;
EXECUTE FUNCTION MQSendClob('MYSERVICE', 'MYPOLICY',
filetoclob("/work/mydata", "client"));
commit;
```

This example sends a CLOB to the WMQ with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *msg_data*: filetoclob("/work/mydata", "client")
- *correl_id*: none

Example 4

```
begin;
EXECUTE FUNCTION MQSendClob('MYSERVICE', 'MYPOLICY',
filetoclob("/work/mydata", "client"), 'TEST3');
commit;
```

This example sends a CLOB to the WMQ with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *msg_data*: filetoclob("/work/mydata", "client")
- *correl_id*: "TEST3"

MQSubscribe() function

The MQSubscribe() function is used to register interest in WMQ messages published on one or more topics.

Syntax

```
MQSUBSCRIBE( [ subscriber_name, [ policy_name, ] ] topic)
```

subscriber_name

Optional parameter. Refers to the value in the **pubsubname** column of the **"informix".mqiservice** table. If *subscriber_name* is not specified, IDS.DEFAULT.SUBSCRIBER is used as the subscriber. The maximum size of *subscriber_name* is 48 bytes.

policy_name

Optional parameter. Refers to the value in the **policyname** column of the **"informix".mqipolicy** table. If *policy_name* is not specified, IDS.DEFAULT.PUB.SUB.POLICY is used as the policy. The maximum size of *policy_name* is 48 bytes.

topic

Required parameter. A string containing the topic for the message publication. The maximum size of a topic is 40 bytes. Multiple topics can be specified in one string (up to 40 characters long). Each topic must be separated by a colon. For example, "t1:t2:the third topic" indicates that the message is associated with all three topics: t1, t2, and the third topic. If no topic is specified, none are associated with the message.

Usage

The MQSubscribe() function is used to register interest in WMQ messages published on a specified topic. The *subscriber_name* specifies a logical destination for messages that match the specified topic. Messages published on the topic are placed on the queue referred by the service pointed to by the **receiver** column for the subscriber (*subscriber_name* parameter). These messages can be read or received through subsequent calls to the MQRead() and MQReceive() functions on the receiver service.

This function requires the installation of the WMQ Publish/Subscribe Component of WMQ and that the Message Broker must be running.

The following table describes how the arguments for the MQSubscribe() function are interpreted.

Table 17. MQSubscribe() argument interpretation

Usage	Argument interpretation
MQSubscribe(arg1)	arg1 = *topic*
MQSubscribe(arg1, arg2)	arg1 = *service_name* arg2 = *topic*
MQSubscribe(arg1, arg2, arg3)	arg1 = *service_name* arg2 = *policy_name* arg3 = *topic*

Return codes

1

The operation was successful.

Error

The operation was unsuccessful.

Examples

Example 1

The following table contains sample rows and columns in the **"informix".mqipubsub** table.

Sample rows	pubsubname column	receiver column	pubsubtype column
Sample row 1	'IDS.DEFAULT. PUBLISHER'	"	'Publisher'
Sample row 2	'IDS.DEFAULT. SUBSCRIBER'	'IDS.DEFAULT. SUBSCRIBER.RECEIVER'	'Subscriber'

```
begin;
EXECUTE FUNCTION MQSubscribe('IDS.DEFAULT.SUBSCRIBER',
   'IDS.DEFAULT.PUB.SUB.POLICY', 'Weather');
commit;
```

This statement demonstrates a subscriber registering an interest in messages containing the topic "Weather" with the following parameters:

- *subscriber_name*: "IDS.DEFAULT.SUBSCRIBER"
- *policy_name*: "IDS.DEFAULT.PUB.SUB.POLICY"
- *topic*: "Weather"

```
begin;
EXECUTE FUNCTION MQPublish('IDS.DEFAULT.PUBLISHER',
'IDS.DEFAULT.PUB.SUB.POLICY', 'Rain', 'Weather');
commit;
```

This statement publishes the message with the following parameters:

- *publisher_name*: "IDS.DEFAULT.PUBLISHER"
- *policy_name*: "IDS.DEFAULT.PUB.SUB.POLICY"
- *msg_data*: "Rain"
- *topic*: "Weather"
- *correl_id*: none

```
begin;
EXECUTE FUNCTION MQReceive('IDS.DEFAULT.SUBSCRIBER.RECEIVER',
'IDS.DEFAULT.PUB.SUB.POLICY');
commit;
```

This statement receives the message with the following parameters (it returns "Rain"):

- *service_name*: "IDS.DEFAULT.SUBSCRIBER.RECEIVER"
- *policy_name*: "IDS.DEFAULT.PUB.SUB.POLICY"

Example 2

```
begin;
EXECUTE FUNCTION MQSubscribe('Weather');
commit;
```

This example demonstrates a subscriber registering an interest in messages containing the topics "Weather" with the following parameters:

- *subscriber_name*: default subscriber
- *policy_name*: default policy
- *topic*: "Weather"

Example 3

```
begin;
EXECUTE FUNCTION MQSubscribe('PORTFOLIO-UPDATES',
 'BASIC-POLICY', 'Stocks:Bonds');
commit;
```

This example demonstrates a subscriber registering an interest in messages containing the topics "Stocks" and "Bonds" with the following parameters:

- *subscriber_name*: "PORTFOLIO-UPDATES"
- *policy_name*: "BASIC-POLICY"
- *topic*: "Stocks", "Bonds"

MQTrace() function

The MQTrace() procedure specifies the level of tracing and the location to which the trace file is written.

Syntax

```
MQTRACE(trace_level, trace_file)
```

trace_level

Required parameter. Integer value specifying the trace level, currently only a value of greater than 50 results in output.

trace_file

Required parameter. The full path and name of the file to which trace information is appended. The file must be writable by user **informix**.

To enable tracing, you must first create a trace class by inserting a record into the **systemtraceclasses** system catalog:

```
insert into informix.systraceclasses(name) values ('idsmq')
```

For more details regarding tracing, see the *HCL OneDB™ Guide to SQL: Reference*.

Example

Enable tracing at a level of 50 with an output file of `/tmp/trace.log`:

```
EXECUTE PROCEDURE MQTrace(50, '/tmp/trace.log');
```

Execute a request:

```
begin;
EXECUTE FUNCTION MQSend('IDS');
commit;
```

Look at the trace output:

```
14:19:38 Trace ON level : 50
14:19:47  >>ENTER : mqSend<<
14:19:47    status:corrid is null
14:19:47  >>ENTER : MqOpen<<
14:19:47   status:MqOpen @ build_get_mq_cache()
14:19:47  >>ENTER : build_get_mq_cache<<
14:19:47   status:build_get_mq_cache @ mi_get_database_info()
14:19:47   status:build_get_mq_cache @ build_mq_service_cache()
14:19:47  >>ENTER : build_mq_service_cache<<
14:19:47  <<EXIT : build_mq_service_cache>>
14:19:47   status:build_get_mq_cache @ build_mq_policy_cache()
14:19:47  >>ENTER : build_mq_policy_cache<<
14:19:47  <<EXIT : build_mq_policy_cache>>
14:19:47   status:build_get_mq_cache @ build_mq_pubsub_cache()
14:19:47  >>ENTER : build_mq_pubsub_cache<<
14:19:47  <<EXIT : build_mq_pubsub_cache>>
14:19:47  <<EXIT : build_get_mq_cache>>
14:19:47   status:MqOpen @ MqiGetServicePolicy()
14:19:47  >>ENTER : MqiGetServicePolicy<<
14:19:47  <<EXIT : MqiGetServicePolicy>>
14:19:47    MQI:MqOpen @ MQCONNX()
14:19:47   status:MqOpen @ MqXadsRegister()
14:19:47  >>ENTER : MqXadsRegister<<
14:19:47   status:MqXadsRegister @ ax_reg()
14:19:47  <<EXIT : MqXadsRegister>>
14:19:47   status:MqOpen @ MqGetMqiContext()
14:19:47  >>ENTER : MqGetMqiContext<<
14:19:47    MQI:MqGetMqiContext @ MQOPEN()
14:19:47  <<EXIT : MqGetMqiContext>>
14:19:47  <<EXIT : MqOpen>>
14:19:47  >>ENTER : MqTransmit<<
14:19:47  >>ENTER : MqBuildMQPMO<<
14:19:47  <<EXIT : MqBuildMQPMO>>
```

```
14:19:47  >>ENTER : MqBuildMQMDSend<<
14:19:47  <<EXIT : MqBuildMQMDSend>>
14:19:47    MQI:MqTransmit @ MQPUT()
14:19:47  <<EXIT : MqTransmit>>
14:19:47  <<EXIT : mqSend>>
14:19:47  >>ENTER : MqEndTran<<
14:19:47    MQI:MqEndTran @ MQCMIT()
14:19:47   status:MqEndTran @ MqShut()
14:19:47  >>ENTER : MqShut<<
14:19:47   status:MqEndTran @ MQDISC
14:19:47  <<EXIT : MqEndTran>>:
```

MQUnsubscribe() function

The MQUnsubscribe() function is used to unregister interest in WMQ messages published on one or more topics.

Syntax

MQUNSUBSCRIBE([*subscriber_name*, [*policy_name*,]] *topic*)

subscriber_name

Optional parameter. Refers to the value in the **pubsubname** column of the **"informix".mqiservice** table. If *subscriber_name* is not specified, IDS.DEFAULT.SUBSCRIBER is used as the subscriber. The maximum size of *subscriber_name* is 48 bytes.

policy_name

Optional parameter. Refers to the value in the **policyname** column of the **"informix".mqipolicy** table. If *policy_name* is not specified, IDS.DEFAULT.PUB.SUB.POLICY is used as the policy. The maximum size of *policy_name* is 48 bytes.

topic

Required parameter. A string containing the topic for the message publication. The maximum size of a topic is 40 bytes. Multiple topics can be specified in one string (up to 40 characters long). Each topic must be separated by a colon. For example, "t1:t2:the third topic" indicates that the message is associated with all three topics: t1, t2, and the third topic. If no topic is specified, none are associated with the message.

Usage

The MQUnsubscribe() function is used to unregister interest in WMQ messages subscription on a specified topic. The *subscriber_name* specifies a logical destination for messages that match the specified topic.

This function requires the installation of the WMQ Publish/Subscribe Component of WMQ and that the Message Broker must be running.

The following table describes how the arguments for the MQUnsubscribe() function are interpreted.

Table 18. MQUnsubscribe() argument interpretation

Usage	Argument interpretation
MQUnsubscribe(arg1)	arg1 = *topic*
MQUnsubscribe(arg1, arg2)	arg1 = *service_name* arg2 = *topic*
MQUnsubscribe(arg1, arg2, arg3)	arg1 = *service_name* arg2 = *policy_name* arg3 = *topic*

Return codes

1

The operation was successful.

Error

The operation was unsuccessful.

Examples

Example 1

```
begin;
EXECUTE FUNCTION MQUnsubscribe('Weather');
commit;
```

This example demonstrates unsubscribing an interest in messages containing the topic "Weather" with the following parameters:

- *subscriber_name*: default subscriber
- *policy_name*: default policy
- *topic*: "Weather"

Example 2

```
begin;
EXECUTE FUNCTION MQUnsubscribe('PORTFOLIO-UPDATES','BASIC-POLICY',
      'Stocks:Bonds');
commit;
```

This example demonstrates unsubscribing an interest in messages containing the topics "Stocks" and "Bonds" with the following parameters:

- *subscriber_name*: "PORTFOLIO-UPDATES"
- *policy_name*: "BASIC-POLICY"
- *topic*: "Stocks", "Bonds"

MQVersion() function

The MQVersion() function returns version information.

The MQVersion() function returns the version of the MQ messaging extension.

Syntax

```
MQVersion()
```

Example

Show the version:

```
EXECUTE FUNCTION MQVersion();
OutPut of the MQVersion() function: MQBLADE 2.0 on 29-MAR-2005
```

MQ messaging configuration parameters

When you use MQ messaging over a network, you must set several database server configuration parameters,

These configuration parameters correspond to (WMQ) environment variables.

MQSERVER configuration parameter

Use the MQSERVER configuration parameter to define a channel, specify the location and specify the communication method to be used.

`onconfig.std` *value*

none

range of values

ChannelName/*TransportType*/*ConnectionName*

takes effect

When the database server is stopped and restarted

Usage

You must set this configuration parameter when you use MQ messaging over a network. The configuration parameter contains the same information as the same as the WMQ MQSERVER environment variable.

The connection name must be a fully qualified network name.

The connection and channel names cannot include contain the forward slash ("�) character, because it is used to separate the channel name, transport type, and connection name.

MQCHLLIB configuration parameter

Use the MQCHLLIB configuration parameter to specify the path to the directory containing the client channel definition table.

`onconfig.std` *value*

none

range of values

complete path name

takes effect

When the database server is stopped and restarted

Usage

You must set this configuration parameter when you use MQ messaging over a network.

For example, if the path is `/var/mqm`, specify:

```
MQCHLLIB /var/mqm
```

MQCHLTAB configuration parameter

Use the MQCHLTAB configuration parameter to specify the name of the client channel definition table.

`onconfig.std` *value*

none

range of values

String for the file name

takes effect

When the database server is stopped and restarted

Usage

You must set this configuration parameter when you use MQ messaging over a network.

The default file name in the WMQ MQCHLTAB environment variable is `AMQCLCHL.TAB`.

For example, if the name of the client channel definition table that you are using is `CCD1`, specify:

```
MQCHLTAB CCD1.TAB
```

MQ messaging error handling

This topic describes MQ messaging error codes.

SQL State	Description
MQ000	Memory allocation failure in %FUNC%.
MQPOL	MQOPEN Policy : %POLICY%
MQSES	MQOPEN Session : %SESSION%
MQRCV	Read %BYTES% from the queue.
MQNMS	No data read/received, queue empty.
MQSUB	Subscribing to %SUBSCRIBE%.
MQVNV	VTI Table definition parameter NAME:%NAME% VALUE:%VALUE%.
MQNPL	VTI No policy defined for table mapped to MQ. Must define table with policy attribute.
MQNSV	VTI No service defined for table mapped to MQ. Must define table with service attribute.
MQNAC	VTI No access defined for table mapped to MQ. Must define table with access attribute.
MQBAC	VTI Invalid Access specification FOUND:%VALUE%, possible values%VALONE% or %VALTWO%.
MQVCN	VTI Qualified : Column 'correlid' cannot be qualified with NULL.
MQVTB	Table missing required 'message' column. Message column is bound to the queue, it is mandatory.
MQVSP	VTI mapped Queue did not include the POLICY and SESSION columns.
MQVIA	VTI table definition invalid access type (%VALUE%), valid access types are %READ% or %RECEIVE%.
MQVMS	VTI mapped queue missing SERVICE specification.
MQVMA	VTI mapped QUEUE creation did not include ACCESS definition.
MQVMP	VTI mapped QUEUE creation did not include POLICY specification.
MQVQC	VTI queue mapping, Column '%COLUMN%' must be qualified with a constant.
MQVQN	VTI queue mapping, Column '%COLUMN%' cannot be qualified with NULL.
MQVQE	VTI queue mapping, Column '%COLUMN%' can only use equality operator.
MQVQF	VTI queue mapping, column '%COLUMN%' - failed to fetch field.
MQSUN	Invalid selector '%IDX%' found, path not possible.
MQERX	Extended error : '%FUNC%', code:%CODE% explain: %EXPLAIN%, refer to MQSeries® publication for further description.
MQGEN	%FUNC% encountered error %ERR% with accompanying message : %MSG%
MQTNL	Topic cannot be NULL.
MQCNL	Internal error encountered NULL context.
MQNLM	Cannot send NULL message.

SQL State	Description
MQVNQ	MQSeries® underlying qualification system does not support negation.
MQVDQ	Qualifications cannot bridge between MQSeries® and database.
MQEDN	MQ Transport error, service '%NAME%' underlying queue manager may not be activated.
MQEPL	Policy '%POLICY%' could not be found in the repository.
MQRLN	Error during read, expected %EXPECT%, received:%READ%.
MQELO	Error attempting to fetch CLOB, function:%NAME% returned %CODE%.
MQRDA	MQ Transport error, service '%NAME%' underlying transpost layer not enabled to receive requests
MQSDA	MQ Transport error, service '%NAME%' underlying transpost layer not enabled to send requests
MQVQM	MQSeries® : Cannot have multiple qualifies for the same column (%COLUMN%).
MQRFQ	Retrieved entries from queue, at least one entry failed qualification - data lost.
MQQCI	Qualification column invalid, only can qualify on 'topic' and 'correlid'.
MQGER	MQ Error : %MSG%
MQGVT	MQ VTI Error : %MSG%
MQZCO	Correlation value found to be zero length, invalid value for MQSeries®.
MQVTN	Must supply name of VTI table.
MQ018	FUNCTION:%NAME%, SERVICE:%SERVICE%, POLICY:%POLICY% :: The specified (sender, receiver, distribution list, publisher, or subscriber) service was not found, so the request was not carried out.
MQ020	FUNCTION:%NAME%, SERVICE:%SERVICE%, POLICY:%POLICY% :: The specified policy was not found, so the request was not carried out.
MQT40	Topic exceeded forty character maximum.
MQINX	Input too large, maximum:%len% found:%txt%
MQITM	Invalid table 'msg' column size %len%, valid range (1-%max%)
MQEXT	AMRC_TRANPORT_ERR, fetched secondary error at:%NAME%, MQI error :%ERR%
MQXAR	Xadatasource (%XADS%) registration error : FUNCTION: %FUNCTION%, RETURN VALUE: %VALUE%
MQ010	FUNCTION:%NAME%: Unable to obtain database information.
MQ011	FUNCTION:%NAME%: Error while querying table:%TABNAME%
MQ012	FUNCTION:%NAME%: Unexpected NULL value while querying the table:%TABNAME%
MQ013	FUNCTION:%NAME%: Unexpected return value from mi function while querying table:%TABNAME%
MQ014	FUNCTION:%NAME%: Unexpected failure opening mi connection while querying table:%TABNAME%

SQL State	Description
MQMQI	FUNCTION:%FNAME%, SERVICE:%SERVICE%, POLICY:%POLICY% :: MQI Error generated by %MQINAME% with CompCode=%CCODE%, Reason=%REASON%.
MQ015	FUNCTION:%FNAME%, SERVICE:%SERVICE%, POLICY:%POLICY% :: %NAME% is not present in the database %TABNAME% table.
MQ016	FUNCTION:%FNAME%, SERVICE:%SERVICE%, POLICY:%POLICY% :: Connection to Multiple QueueManagers are not allowed in the same transaction.
MQ019	FUNCTION:%FNAME%, SERVICE:%SERVICE%, POLICY:%POLICY% :: Internal Error. not able to switch to the virtual processor where the MQCONNX() is invoked.
MQ017	FUNCTION:%FNAME%, SERVICE:%SERVICE%, POLICY:%POLICY% :: Internal Error. The Virtual processor class not the same as ""MQ""

Sample MQ messaging code

This topic contains sample SQL statements that you can run in the **stores_demo** database, using DB-Access.

The sample statements are for one queue manager. However, you can use multiple queue managers.

```
begin;

select MQSEND ('lser.qm1', 'IDS.DEFAULT.POLICY',
       TRIM(fname)    || '  ' || TRIM(lname) || '|'
       || TRIM(company)  || ', ' || TRIM(address1) || ', '
       || TRIM(NVL(address2,"")) || ', ' || TRIM(city) || ', '
       || state || ', ' || zipcode || '|' || TRIM(phone) || '|' ,
      state)
     from customer;

select MQSEND ('lser.qm1', 'IDS.DEFAULT.POLICY',
       stock_num || '|' || manu_code || '|' || TRIM(description)
       || '|' || unit_price || '|' || unit
       || '|' || TRIM(unit_descr) || '|' ,
       manu_code)
     from stock;
commit;



select first 3 MQREAD('lser.qm1') from systables;
begin;
execute function MQREAD('lser.qm1','IDS.DEFAULT.POLICY','AZ');
rollback;

begin;
execute function MQREAD ('lser.qm1');
execute function MQREAD ('lser.qm1');
execute function MQRECEIVE ('lser.qm1');
execute function MQRECEIVE ('lser.qm1');
rollback;

begin;
```

```
select first 5 MQREAD ('lser.qm1') from systables;
select first 5 MQREAD ('lser.qm1') from systables;
select first 1 MQRECEIVE ('lser.qm1','IDS.DEFAULT.POLICY','AZ')
 from systables;
select first 1 MQRECEIVE ('lser.qm1','IDS.DEFAULT.POLICY','HSK')
 from systables;
rollback;

begin;
select first 5 MQREAD ('lser.qm1') from systables;
select first 5 MQREAD ('lser.qm1') from systables;
select first 1 MQRECEIVE ('lser.qm1','IDS.DEFAULT.POLICY','AZ')
 from systables;
select first 1 MQRECEIVE ('lser.qm1','IDS.DEFAULT.POLICY','HSK')
 from systables;
commit;

execute function mqinquire('lser.qm1',20);
execute function mqinquire('lser.qm1',"MQIA_Q_TYPE");
execute function mqinquire('lser.qm1',3);
execute function mqinquire('lser.qm1',"MQIA_CURRENT_Q_DEPTH");
execute function mqhasmessage('lser.qm1');
execute function mqhasmessage('lser.qm1','IDS.DEFAULT.POLICY','CA');
execute function mqhasmessage('lser.qm1','IDS.DEFAULT.POLICY','XY');

execute function MQCreateVtiRead("qm0vti", "lser.qm1");
execute function MQCreateVtiReceive("qm0vtir", "lser.qm1");
execute function MQCreateVtiWrite("qm0vtiw", "lser.qm1");
execute function MQCreateVtiReceive("qm1vti", "lser.qm1");

insert into qm0vtiw(msg) values ("Informix Dynamic Server");
begin;

select skip 10 first 5 * from qm0vtir;
select * from qm1vti;
insert into qm1vti(msg) values ("Informix Dynamic Server");
select * from qm1vti;
commit;
```

Binary data types

The binary18 and binaryvar data types allow you to store binary-encoded strings, which can be indexed for quick retrieval.

You can use string manipulation functions to validate the data types and bitwise operation functions that allow you to perform bitwise logical AND, OR, XOR comparisons or apply a bitwise logical NOT to a string.

Because the binary data types are unstructured types, they can store many different types of information, for example, IP addresses, MAC addresses, or device identification numbers from RFID tags. The binary data types can also store encrypted data in binary format, which saves disk space. Instead of storing an IP address like *xxx.xxx.xxx.xxx* as a CHAR(15) data type, you can store it as a binaryvar data type, which uses only 6 bytes.

Binary data types overview

To implement binary data types, the Scheduler must be running and the database must conform to requirements. The binary18 and binaryvar data types have certain restrictions due to the nature of binary data.

The Scheduler must be running in the database server. If the Scheduler is not running when you create a binary data type, a message that the data type is not found is returned.

The database that contains the binary data types must meet the following requirements:

- The database must be logged.
- The database must not be defined as an ANSI database.

If you attempt to create a binary data type in an unlogged or ANSI database, the message `DataBlade registration failed` is printed in the online message log.

Binary data type can be used in the following situations:

- The binary data types are allowed in Enterprise Replication.
- Casts to and from the LVARCHAR data type are allowed as are implicit casts between the binary18 and binaryvar data types.
- The aggregate functions **COUNT DISTINCT()**, **DISTINCT()**, **MAX()**, and **MIN()** are supported.

Binary data types have the following limitations:

- The only arithmetic operations that are supported are the bitwise operators: **bit_and()**, **bit_or()**, **bit_xor()**, and **bit_complement()**.
- The LIKE and MATCHES conditions are not supported.

Store and index binary data

This chapter describes the binary data types and how to insert and index binary data.

Binary data types

You can store and index binary data by using the binaryvar and binary18 data types.

The binaryvar data type

The binaryvar data type is a variable-length opaque type with a maximum length of 255 bytes.

The binary18 data type

The binary18 data type is a fixed-length opaque data type that holds 18 bytes. Input strings shorter than 18 bytes are right-padded with zeros (`00`). Strings longer than 18 bytes are truncated.

The binary18 data type has the advantage of not having its length stored as part of the byte stream. When inserting data into the binaryvar data type, the first byte must be the length of the byte array. The binary18 data type does not have this restriction.

ASCII representation of binary data types

Binary data types are input using a 2-digit ASCII representation of the characters in the hexadecimal range of 0-9, A-F. The characters A-F are not case-sensitive and you can add a leading **0x** prefix to the string. You must enter an even number of bytes up to the maximum number of encoded bytes permitted, otherwise an error is generated. For example, 36 bytes are input to represent the binary18 data type. No spaces or other separators are supported.

Each 2-byte increment of the input string is stored as a single byte. For example, the 2-byte ASCII representation of "AB" in hexadecimal notation is divided into blocks of four binary characters, where `1010` `1011` equals one byte.

Binary data type examples

Example 1: binaryvar data type

The following code stores the binary string of `0123456789` on disk:

```
CREATE TABLE  bindata_test (int_col integer, bin_col binaryvar)

INSERT INTO bindata_test values (1, '30313233343536373839')
INSERT INTO bindata_test values (2, '0X30313233343536373839')
```

Example 2: binary18 data type

The following code inserts the string IBMCORPORATION2006:

```
CREATE TABLE bindata_test (int_col integer, bin_col binary18)

INSERT INTO bindata_test values (1,'49424d434f52504f524154494f4e32303036')
INSERT INTO bindata_test values (2,'0x49424d434f52504f524154494f3e32303036')
```

Insert binary data

You can use one of two methods to insert binary data with the binary data types: an SQL INSERT statement that uses the ASCII representation of the binary data type or an SQL INSERT statement from a Java™ or C program that treats the column as a byte stream. For example, given the following table:

```
CREATE TABLE network_table (
mac_address binaryvar NOT NULL,
device_name varchar(128),
device_location varchar(128),
device_ip_address binaryvar,
date_purchased date,
last_serviced date)
```

Using an SQL INSERT statement that uses the ASCII representation of the binaryvar or binary18 column:

```
INSERT INTO network_table VALUES ( '000012DF4F6C', 'Network Router 1',
'Basement',  'C0A80042', '01/01/2001', '01/01/2006');
```

Using an SQL INSERT statement from a Java™ program that treats the column as a byte stream, such as the JDBC **setBytes()** method:

```
String binsqlstmt = "INSERT INTO network_table (mac_address, device_name,
device_location, device_ip_address) VALUES ( ?, ?, ?, ? );
PreparedStatement stmt = null;
byte[] maddr = new byte[6];
byte[] ipaddr = new byte[4];
try
{
   stmt = conn.prepareStatement(binsqlstmt);
   maddr[0] = 0;
    maddr[1] = 0;
   maddr[2] = 18;
   maddr[3] = -33;
   maddr[4] = 79;
   maddr[5] = 108;
   stmt.setBytes(1, maddr);
   stmt.setString(2, "Network Router 1");
   stmt.setString(3, "Basement");
   ipaddr[0] = -64;
   ipaddr[1] = -88;
   ipaddr[2] = 0;
   ipaddr[3] = 66;
   stmt.setBytes(4,ipaddr);
   stmt.executeUpdate();
   stmt.close()
}
catch
{
   System.out.println("Exception: " + e);
   e.printStackTrace(System.out);
   throw e;
}
```

Index binary data

The binaryvar and binary18 data types support indexing using the B-tree access method for single-column indexes and composite indexes. Nested-loop join operations are also supported.

For example, given the following table:

```
CREATE TABLE network_table (
mac_address binaryvar NOT NULL,
device_name varchar(128),
device_location varchar(128),
device_ip_address binaryvar,
date_purchased date,
last_serviced date)
```

The following statement can be used to create the index:

```
CREATE UNIQUE INDEX netmac_pk ON network_table (mac_address) USING btree;
```

Binary data type functions

This chapter describes functions for the binary data types and provides detailed information about each function's syntax and usage.

Bitwise operation functions

These functions perform bitwise operations on binary18 or binaryvar fields. The expressions can be either binary18 or binaryvar columns or they can be expressions that have been implicitly or explicitly cast to either the binary18 or the binaryvar data type.

The return type for all of these functions is either the binary18 or the binaryvar data type.

The bit_and() function

The bit_and() function performs a bitwise logical AND operation on two binary data type columns.

Syntax

```
bit_and(column1, column2)
```

column1*, *column2

Two input binary data type columns.

Usage

If the columns are different lengths, the return value is the same length as the longer input parameter with the logical AND operation performed up to the length of the shorter parameter.

Return codes

The function returns the value of the bitwise logical AND operation.

If either parameter is NULL, the return value is also NULL.

Example

In the following example, the value of `binaryvar_col1` is '00086000'.

```
SELECT bit_and(binaryvar_col1, '0003C000'::binaryvar) FROM table WHERE x = 1
expression
------------------
00004000
```

The bit_complement() function

The bit_complement() function performs a logical NOT, or *one's complement* on a single binary data type column.

Syntax

```
bit_complement(column)
```

column

The input binary data type column.

Usage

The function changes each binary digit to its complement. Each `0` becomes a `1` and each `1` becomes a `0`.

Return codes

The function returns the value of the bitwise logical NOT operation.

Example

In the following example the value of `binaryvarcol1` is '00086000':

```
SELECT bit_complement(binaryvar_col1) FROM table WHERE x = 1
expression
------------------
FFF79FFF
```

The bit_or() function

The bit_or() function performs a bitwise logical OR on two binary data type columns.

Syntax

```
bit_or(column1, column2)
```

column1*, *column2

Two input binary data type columns.

Usage

If the columns are of different length, the return value is the same length as the longer input parameter, with the OR operation performed up to the length of the shorter parameter. The remainder of the return value is the unprocessed data in the longer string.

Return codes

The function returns the value of the bitwise logical OR operation.

If either parameter is NULL, the return value is also NULL.

Example

In the following example, the value `binaryvarcol1` is '00006000':

```
SELECT bit_or(binaryvar_col1, '00080000'::binaryvar) FROM table WHERE x = 1
expression
------------------
00086000
```

The bit_xor() function

The bit_xor() function performs a bitwise logical XOR on two binary data type columns.

Syntax

```
bit_xor(column1, column2)
```

column1*, *column2

Two input binary data type columns.

Usage

If the columns are of different lengths, the return value is the same length as the longer input parameter, with the XOR operation performed up to the length of the shorter parameter. The remainder of the return value is the unprocessed data in the longer parameter.

Return codes

The function returns the value of the bitwise logical XOR operation.

If either parameter is NULL, the return value is also NULL.

Example

In the following example, the value of `binaryvarcol1` is '00086000':

```
SELECT bit_xor(binaryvar_col1, '00004000'::binaryvar) FROM table WHERE x = 1'
expression
------------------
00082000
```

Support functions for binary data types

Supporting functions for binary data types include the SQL LENGTH() and OCTET_LENGTH() functions that allow you to determine the length of a column. The bdttrace() function is used to trace events related to using binary data types.

The bdtrelease() function

The bdtrelease() function provides the version number of the binary data types.

Syntax

```
bdtrelease(void)
```

Usage

Use the bdtrelease() function when directed to do so by the HCL Software support representative.

Return codes

This function returns the name and version number of the binary data types.

The bdttrace() function

The bdttrace() function specifies the location where the trace file is written.

Syntax

```
bdttrace(filename)
```

filename

The full path and name of the file to which trace information is appended. The file must be writable by user **informix**. If no file name is provided, a standard `session_id.trc` file is placed in the `$ONEDB_HOME/tmp` directory. If the file already exists, the trace information is appended to the file.

Usage

Use the bdttrace() function to troubleshoot events related to binary data types.

To enable tracing, create a trace class by inserting a record into the **systemtraceclasses** system catalog:

```
insert into informix.systraceclasses(name) values ('binaryUDT')
```

For more details regarding tracing, see the *HCL OneDB™ Guide to SQL: Reference*.

Example

```
bdttrace(tracefile)
```

The LENGTH() function

Use the LENGTH() SQL function to determine if the string is from a binaryvar or a binary18 column. The LENGTH() function returns the number of bytes in a column.

Syntax

```
LENGTH(column)
```

column

The binary data type column.

Usage

This function returns the length of the column in bytes as an integer. For the binary18 data type, the function always returns `18`.

For binary data types, the SQL LENGTH() and OCTET_LENGTH() functions return the same value. For more information about length functions, see the *HCL OneDB™ Guide to SQL: Reference*.

Example

```
SELECT length(binaryvar_col) FROM table WHERE binaryvar_col = '0A010204'
expression
------------------
 4
```

The OCTET_LENGTH() function

Use the OCTET_LENGTH() SQL function to determine if the string is from a binaryvar or a binary18 column. The OCTET_LENGTH() function returns the number of octets (bytes).

Syntax

```
OCTET_LENGTH(column)
```

column

The binary data type column.

Usage

This function returns the length of the column in bytes as an integer. For the binary18 data type, the function always returns `18`.

For binary data types, the SQL LENGTH() and OCTET_LENGTH() functions return the same value. For more information about length functions, see the *HCL OneDB™ Guide to SQL: Reference*.

Example

```
SELECT octet_length(binaryvar_col) FROM table WHERE binaryvar_col = '93FB'
expression
------------------
 2
```

Basic Text Search

You can perform basic text searching for words and phrases in a document repository stored in a column of a table.

In traditional relational database systems, you must use a LIKE or MATCHES condition to search for text data and use the database server to perform the search. HCL OneDB™ uses the open source CLucene text search package to perform basic text searches. This text search package and its associated functions, known as the text search *engine*, is specifically designed to perform fast retrieval and automatic indexing of text data. The text search engine runs in virtual processors that are controlled by the database server.

To perform basic text searches, you create a **bts** index on one or more text columns and then use the bts_contains() search predicate function to query the text data.

You can configure how to index the text data by specifying an analyzer. Each analyzer uses different criteria to index the data. By default the Standard analyzer is used.

You can specify synonyms for data that has multiple words for the same information, for example, proper names with multiple spellings. You can use canonical mapping to create a static list of synonyms. You can create a thesaurus with synonyms that you can update dynamically.

To search for words and phrases you use a predicate called bts_contains() that instructs the database server to call the text search engine to perform the search.

For example, to search for the string `century` in the column **brands** in the table **products** you use the following statement:

```
SELECT id FROM products
WHERE bts_contains(brands, 'century');
```

The search predicate takes a variety of arguments to make the search more detailed than one using a LIKE condition. Search strategies include single and multiple character wildcard searches, fuzzy and proximity searches, AND, OR and NOT Boolean operations, range options, and term-boosting.

If you store XML, JSON, or BSON documents, you can create customized structured indexes so that you can search columns by XML tags, attributes, and paths, or JSON fields, values, and paths. Customize the index with XML or JSON index parameters.

You can search for unstructured text or, if you use XML index parameters, you can search columns with XML documents by tags, attributes, or XML paths.

You can use basic text search functions to perform maintenance tasks, such as compacting the **bts** index and obtaining the list of indexed field names.

Preparing for basic text searching

Before you can perform basic text searching, you must prepare the server environment and create the **bts** index. Review the requirements and restrictions.

About this task

To prepare for basic text searching, complete these tasks:

1. Create a default sbspace.
2. **Optional:** Create an sbspace for the **bts** index.
3. **Optional:** Create a space for temporary data.
4. Create the **bts** index.

What to do next

Basic text search functions run in a BTS virtual processor, which means that only one query or other type of index operation runs at a time in each virtual processor. When you create a bts index, the BTS virtual processor class is created automatically.

Basic text search requirements and restrictions

When you plan how to configure basic text searching, you must understand the requirements and restrictions.

Database server requirement

The Scheduler must be running in the database server. If the Scheduler is not running when you create a **bts** index, a message that the access method is not found is returned.

Database requirements

The database that contains the **bts** index must be logged and must not be an ANSI database. If you attempt to create a **bts** index in an unlogged or ANSI database, the message `DataBlade registration failed` is printed in the database server message log.

Data type support

To use basic text searching, you must store the text data in a column of data type BLOB, BSON, CHAR, CLOB, JSON, LVARCHAR, NCHAR, NVARCHAR, or VARCHAR.

To use basic text searching, you must store the text data in a column of data type BLOB, CHAR, CLOB, LVARCHAR, NCHAR, NVARCHAR, or VARCHAR.

Although you can store searchable text in a column of the BLOB data type, you cannot create a basic text search index on binary data. BLOB data type columns must contain text.

Locales and languages support

Basic text search queries can use most multi-byte character sets and global language support, including UTF-8, and can use ideographic languages such as Chinese, Korean, and Japanese if you specify the CJK analyzer.

Important: If you use UTF-8 character encoding, including the Chinese GB18030-2000 code set, you must set the GL_USEGLU environment variable before you create the database.

JSON or BSON documents must be in a database with a UTF-8 locale.

High availability support

You can run basic text search queries on primary and all types of secondary servers in high-availability clusters.

Index characteristics restrictions

The following characteristics are not supported for **bts** indexes:

- Fill factors
- Index clustering
- Unique indexes

Indexed document restrictions

If your documents are over 32 KB, store them in columns of type BLOB or CLOB.

The size of a document that you want to index is limited by the amount of available virtual memory on your machine. For example, if you have 1 GB of available virtual memory, you can only index documents that are smaller than 1 GB.

Query restrictions

You cannot include basic text search queries in distributed queries or parallel database queries.

Creating a default sbspace

A default sbspace must exist before you create a **bts** index. The database server sets up internal directories for basic text searching in a default sbspace.

About this task

The database server also stores **bts** indexes in the default sbspace unless you explicitly specify another sbspace when you create the index. Be sure the default sbspace is large enough to hold all of these objects. Monitor the size of the default sbspace and increase its size when necessary.

If you do not explicitly create a default sbspace and set the SBSPACENAME configuration parameter in the `onconfig` file before you create a **bts** index, the database server creates a default sbspace automatically before running the CREATE INDEX statement, according to the following criteria in this order:

- If storage provisioning is configured, the default sbspace is created in the designated storage pool.
- If the root dbspace was created in a directory, the default sbspace is created in the same directory and could use the same files system as the root dbspace.
- If the root dbspace is a raw device in the `/dev` directory, the default sbspace is created in the `$ONEDB_HOME/tmp` directory.

The sbspace for **bts** index must have buffering enabled. Buffering is enabled by default when you create an sbspace. You can use various methods to create an sbspace, including the onspaces utility, the SQL administration API task() function with the **create sbspace** argument, or through storage provisioning, if you have configured a storage pool.

To create the default sbspace:

1. Set the SBSPACENAME configuration parameter in the configuration file to the name of your default sbspace.
   Example

   The following example sets the name of the default sbspace to **sbsp1**:

   ```
   SBSPACENAME sbsp1
   ```
2. Restart the database server.
3. Create the sbspace.
   Example

   The following example creates an sbspace called **sbsp1** in the file `c:\IFMXDATA\sbspace` by using the onspaces utility:

   ```
   onspaces -c -S sbsp1 -p c:\IFMXDATA\sbspace -o 0 -s 100000
   ```

Creating a space for the bts index

Each **bts** index is stored in one or more sbspaces. You can create a dedicated sbspace to store your **bts** index and then specify that sbspace name when you create the **bts** index. For backwards compatibility, you can continue to store **bts** indexes in extspaces.

About this task

If you do not create a separate sbspace for your **bts** indexes, the database server stores **bts** indexes in the default sbspace.

In general, the sbspace for a **bts** index should be at least the size of the data being indexed. A highly optimized index might take up to three times the size of the data being indexed.

The sbspace for **bts** index must have buffering enabled. Buffering is enabled by default when you create an sbspace. You can use various methods to create an sbspace, including the onspaces utility, the SQL administration API task() function with the **create sbspace** argument, or through storage provisioning, if you have configured a storage pool.

To create an sbspace, use the onspaces utility. For example:

```
onspaces -c -S bts_sbspace -o 0 -s 100000 -p /dev/sbspace
```

To create an extspace:

1. Create a directory for the index.
2. Create the extspace by using the onspaces utility.

Example

The following example creates a directory and an extspace:

```
mkdir bts_extspace_directory
onspaces -c -x bts_extspace -l "/bts_extspace_directory"
```

Creating a space for temporary data

Basic text searching creates temporary data while processing **bts** indexes. You can create a separate space for temporary data and specify it when you create the **bts** index.

Before you begin

For best performance, the space should be a temporary sbspace since data and metadata for temporary files are not logged. However, you can also use an sbspace or an extspace.

If you do not specify a separate space for temporary data when you create the **bts** index with the tempspace index parameter, the database server stores temporary data in one of the following locations, according to the criteria in the following order:

- The sbspace specified in the CREATE INDEX statement.
- A temporary sbspace that is specified by the SBSPACETEMP configuration parameter. The temporary sbspace with the most free space is used. If no temporary sbspaces are listed, the sbspace with the most free space is used.
- If the SBSPACETEMP configuration parameter is not set and you have a storage pool that is set up, a temporary sbspace is created and the SBSPACETEMP configuration parameter is set dynamically in the `onconfig` file.
- The sbspace specified by the SBSPACENAME configuration parameter.

About this task

To create a temporary sbspace, use the onspaces utility with the **-t** option. (Do not include the **-Df "LOGGING=ON"** option.)

Example

For example:

```
onspaces -c -S temp_sbspace -t -o 0 -s 50000 -p /dev/temp_sbspace
```

What to do next

Alternatively, you could create a temporary sbspace through storage provisioning, if you have configured a storage pool.

Creating a bts index

You create a **bts** index by using the **bts** access method and specifying index parameters and other options.

About this task

Before you create a **bts** index, plan which index parameters and other options you want to use.

To create a **bts** index:

1. Complete prerequisite tasks that are necessary for the index parameters that you plan to include for the index. For example, many index parameters use tables or files that you must create before you create the index.
2. Create an index by specifying the **bts** access method.

bts access method syntax

The **bts** access method is a secondary access method to create indexes that support basic text search queries.

Instead of using the **bts** access method to create a **bts** index, you can run the HCL OneDB™ JSON createTextIndex command. Use the same syntax for **bts** index parameters for both methods.

Syntax

CREATE INDEX *index_name* **ON** *table_name* (*column_name* *op_class*) **USING bts** [(<bts index parameters>)] [{ **IN** *space_name* | **FRAGMENT BY EXPRESSION** (*expression*) **IN** *space_name* [**REMAINDER IN** *space_name*] }] ;

bts index parameters

" { | <analyzer index parameter>(explicit id) | <canonical_map index parameter>(explicit id) | **delete=** { **"deferred"** | **"immediate"** }(explicit id) | **field_token_max="***number_tokens***"**(explicit id) | **max_clause_count="***max_clauses***"**(explicit id) | **query_default_field="** { *field* | * } **"**(explicit id) | **query_default_operator=** { **"OR"** | **"AND"** } | **query_log=** { **"no"** | **"yes"** }(explicit id) | <stopwords index parameter>(explicit id) | **tempspace=***tempspace_name*(explicit id) | <thesaurus index parameters>(explicit id) | <xact_memory index parameter>(explicit id) | **xact_ramdirectory="** { **no** | **yes** } **"**(explicit id) | { <XML index parameters>(explicit id) | <JSON index parameters>(explicit id) } }"

Element	Description
column_name	The name of the column in the table that contains the text documents to search.
expression	The expression that defines an index fragment. The expression must return a Boolean value. The expression can contain only columns from the current table and data values from only a single row. The expression cannot include the following elements: • Subqueries • Aggregates are not allowed. T • The built-in CURRENT, DATE, SYSDATE, and TODAY functions • The bts_contains() search predicate For more information about expressions, see Expression on page .
field	The name of the field to set as the default field in basic text search queries instead of the `contents` field.
index_name	The name of the **bts** index.
max_clauses	The maximum number of clauses in a basic text search query. Default is 1024.
number_tokens	The maximum number of tokens to index for each document. Default is 10 000. Maximum is 2 000 000 000.
op_class	The operator class for the data type that is specified in the *column_name* element.
space_name	The name of the sbspace or extspace in which to store the **bts** index.
table_name	The name of the table for which you are creating the index.
tempspace_name	The name of the space in which to store temporary files.

Usage

Include a comma between index parameters.

You must create a **bts** index for each text column that you plan to search. You can either create a separate **bts** index for each text column, or create a composite index on multiple text columns in a table by including multiple column and operator class pairs. You cannot create a composite index that includes a JSON or BSON column. If you want to index each column separately, include the query_default_field="*" index parameter.

You cannot alter the characteristics of a **bts** index after you create it. Instead, you must drop the index and re-create it.

When you create a **bts** index, you specify the operator class that is defined for the data type of the column that is indexed. An operator class is a set of functions that the database server associates with the **bts** access method to optimize queries and build indexes. Each of the data types that support a **bts** index has a corresponding operator class. The following table lists each data type and its corresponding operator class.

Table 19. Data types and the corresponding operator classes

Data type	Operator class
BLOB	bts_blob_ops
BSON	bts_bson_ops
CHAR	bts_char_ops
CLOB	bts_clob_ops
JSON	bts_json_ops
LVARCHAR	bts_lvarchar_ops
NCHAR	bts_nchar_ops
NVARCHAR	bts_nvarchar_ops
VARCHAR	bts_varchar_ops

Examples

Example 1: Create a bts index and store it in an sbspace

For example, suppose that your search data is contained in a column that is named **brands**, of data type CHAR, in a **products** table. To create a **bts** index that is named **desc_idx** in the sbspace **sbsp1**, use the following syntax:

```
CREATE INDEX desc_idx ON products (brands  bts_char_ops)
 USING bts IN sbsp1;
```

Example 2: Create a fragmented bts index

The following example stores the **bts_idx** index in three sbspaces by fragmenting the index according to an expression:

```
CREATE INDEX bts_idx ON bts_tab(col2 bts_char_ops) USING bts
      FRAGMENT BY EXPRESSION
        (col1 <= 1000000) IN bts_sbspace00,
        (col1 > 1000000 and col1 <= 2000000)
        IN bts_sbspace01,
        REMAINDER IN bts_sbspace36;
```

delete index parameter

The delete index parameter controls the optimizing, or compacting, of the index. Optimizing the index removes index information for deleted documents and releases disk space. You can optimize the **bts** index manually, which is the default mode, or automatically.

Optimize the index manually

When you create a **bts** index, the default mode for deleting rows is deferred (delete="deferred"). A delete operation on a row in a table marks the row as deleted in the **bts** index. The disk space can be reclaimed as more documents are added to the index. Queries that are run against **bts** columns do not return the deleted documents.

To release disk space that is occupied by the deleted documents in the index, use the oncheck utility in the format:

```
oncheck -ci -y db_name:table_name#index_name
```

Alternatively, you can use the bts_index_compact() function to release disk space for the rows marked for deletion. The difference between the two methods is that the bts_index_compact() function requires that you know the directory path to the **bts** index, whereas the oncheck utility requires that you know the database name, table name, and the index name. Both methods have the same result.

Delete operations are faster in the deferred mode. The deferred deletion mode is best for large indexes that are updated frequently.

Optimize the index automatically

You can override the deferred deletion mode by creating the **bts** index with the delete="immediate" parameter. In the immediate deletion mode, index information for deleted documents is physically removed from the index after every delete operation. This mode frees up space in the index immediately. However, the immediate deletion mode rewrites the index each time an index entry is deleted, which slows down delete operations and makes the index unusable during the delete operation.

field_token_max index parameter

If you have large documents, you can increase the maximum number of tokens that are indexed by setting the field_token_max index parameter to a positive integer up to 2 000 000 000. By default, 10 000 tokens are indexed in a document. If the average word has 5 characters, approximately 50-60 KB of the document is indexed.

Example

Example

For example, the following statement creates a **bts** index that creates up to 500 000 tokens per document:

```
CREATE INDEX books_bts ON books(book_data bts_lvarchar_ops)
USING bts(field_token_max="500000") IN bts_sbspace;
```

max_clause_count index parameter

You can increase the maximum number of query results by setting the max_clause_count index parameter to a value greater than the default value of 1024.

Basic text queries fail when the maximum number of results is exceeded. If a query results in more than the maximum number of results, you receive the following error:

```
(BTSB0) - bts clucene error: Too Many Clauses
```

This error can occur during a wildcard or fuzzy search.

The limit of results controls virtual memory usage. Queries with large result sets can result in slower performance and the allocation of more virtual segments. You can monitor the number of virtual segments with the onstat -g seg command.

Example

Example

The following statement creates a **bts** index with a maximum number of 4000 query results:

```
CREATE INDEX bts_idx ON bts_tab(text bts_char_ops)
USING bts (max_clause_count="4000")
IN sbspace1;
```

query_default_field index parameter

Set the query_default_field index parameter to a column name to override the default field for basic text search queries from the `contents` field. Set the query_default_field index parameter to * to separately index each column in a composite index.

You do not need to specify the default field in a basic text search query. If you have a structured index on JSON or XML data, you can change the default field to one of the indexed field names or tags.

You do not need to specify the default field in a basic text search query. If you have a structured index on XML data, you can change the default field to one of the indexed tags.

You can create a composite **bts** index on multiple text columns. By default, columns are concatenated and indexed as a single string in the `contents` field. Regardless of which column name you specify in the query, the matching text from all the indexed columns is returned. You can use the query_default_field="*" index parameter to index each column separately so that you can query by column name, which becomes the index field name. When you use the query_default_field="*" index parameter, only the matching text from the column name that you specify in the query is returned. You query multiple columns by including their field names in the format *fieldname*:*string*.

You cannot create a composite index on JSON or BSON columns.

If you combine the query_default_field="*" index parameter with the xmltags index parameter, the composite index is created on only the XML columns.

Example

Examples: Create composite indexes

The following examples use a table with the following structure:

```
CREATE TABLE address(
    fname      char(32),
    lname      char(32),
    address1   varchar(64),
    address2   varchar(64),
    city       char(32),
    province   char(32),
```

```
    country    char(32),
    postalcode char(10)
);
```

You can create a composite **bts** index on multiple columns in the **address** table by using the following statement, which matches each column data type with its corresponding operator class:

```
CREATE INDEX  bts_idx ON address(
    fname      bts_char_ops,
    lname      bts_char_ops,
    address1   bts_varchar_ops,
    address2   bts_varchar_ops,
    city       bts_char_ops,
    province   bts_char_ops,
    country    bts_char_ops,
    postalcode bts_char_ops) USING bts;
```

The resulting composite index concatenates all the columns into the `contents` field. The following two queries would produce the same results because the text is not indexed by column name:

```
SELECT * FROM address WHERE bts_contains(fname, 'john');
SELECT * FROM address WHERE bts_contains(address1, 'john');
```

Alternatively, you can create a composite **bts** index and specify that each column is indexed separately by including the query_default_field="*" index parameter:

```
CREATE INDEX  bts_idx ON address(
    fname      bts_char_ops,
    lname      bts_char_ops,
    address1   bts_varchar_ops,
    address2   bts_varchar_ops,
    city       bts_char_ops,
    province   bts_char_ops,
    country    bts_char_ops,
    postalcode bts_char_ops) USING bts (query_default_field="*");
```

The resulting composite index includes the column name with the indexed text. The following two queries would produce different results:

```
SELECT * FROM address WHERE bts_contains(fname, 'john');
SELECT * FROM address WHERE bts_contains(address1, 'john');
```

The first query finds matches for `john` in the **fname** column and the second query finds matches for `john` in the **address1** column.

The following example searches for a row that contains specific text in two of its columns:

```
SELECT * FROM address WHERE bts_contains(fname, 'john AND city:nipigon');
```

This query returns the rows that contain both `john` in the **fname** column and `nipigon` in the **city** column.

query_log index parameter

You can determine the frequency of queries that are run against a **bts** index by logging queries.

About this task

When tracking is enabled, each query that is run against the **bts** index produces a log record in the `$ONEDB_HOME/tmp/bts_query.log` file. Each log record has five fields, which are separated by pipe characters (|):

query time stamp|*index name*|*partn*|*query*|*number of rows*|

The fields are described in the following table.

Table 20. Query tracking fields

Field name	Data type	Description
query time stamp	DATETIME YEAR TO FRACTION	The time when the query was run.
index name	LVARCHAR	The name of the index.
partn	INTEGER	The identifying code of the physical location of the fragment in which the index is located.
query	LVARCHAR	The syntax of the query.
number of rows	INTEGER	The number of rows that are returned by the query.

You can view the log records by loading them into a table and then querying the table.

This example shows how to track queries.

1. Create the **bts** index with tracking enabled:
   Example
   ```
   CREATE INDEX bts_idx ON products (brands  bts_char_ops)
    USING bts (query_log="yes") IN sbsp1;
   ```
2. Create a table to hold the log records:
   Example
   ```
   CREATE TABLE bts_query_log_data(
     qwhen DATETIME YEAR TO FRACTION,
     idx_name LVARCHAR,
     partn INTEGER,
     query LVARCHAR,
     rows INTEGER);
   ```
3. Load the log records into the log table:
   Example
   ```
   LOAD FROM '$ONEDB_HOME/tmp/bts_query.log' INSERT INTO bts_query_log_data;
   ```
4. Query the log table to view the log records:
   Example

```
SELECT ids_name,query,rows FROM bts_query_log_data;

idx_name bts_idx
query melville
rows 14

idx_name bts_idx
query dickens
rows 29

idx_name bts_idx
query austen
rows 3

3 row(s) retrieved.
```

stopwords index parameter

When you specify a customized stopword list, it replaces the default stopword list. You create a customized stopword list with the **stopwords** index parameter when you create the **bts** index.

stopwords index parameter

" **stopwords="** { ([*field*:] [*word*]) | **file:***directory/filename* | **table:***table.column* } **"** "

Element	Description
column	The name of the column that contains stopwords.
directory	The path for the stopwords file.
field	The XML tag, path, or the column name that is indexed.
filename	The name of the file that contains stopwords.
table	The name of the table that contains stopwords.
word	The term to use as a stopword. Stopwords must be lowercase.

Usage

You can create a stopword list for all fields or customized stopword lists for specific fields. Any words that are listed before any field names become the default stopword list, which is used for all fields not explicitly listed. All words that are listed after a field name and before the next field name are stopwords for the preceding field only. If a field is listed without any words following it, that field does not have a stopword list.

You can specify the list of stopwords in a table column or in a file. The file or table must be readable by the user who is creating the index. Separate the field name and stopword pairs in the file or table by commas, white spaces, new lines, or a

combination of those separators. The file or table becomes read-only when the index is created. If you want to add or change stopword assignments, you must drop and re-create the index.

Examples

Example 1: Input stopwords as inline comma-separated words

Inline comma-separated words are useful when you have only a few stopwords. The following example prevents searching the words "am,", "be," and "are":

```
stopwords="(am,be,are)"
```

The following example shows how to create a **bts** index with an inline comma-separated customized stopword list:

```
CREATE INDEX books_bts ON books(book_data bts_lvarchar_ops)
USING bts(stopwords="(am,be,are)") IN bts_sbspace;
```

Example 2: Input stopwords from a file or a table column

The following example shows the contents of a stopword file where stopwords are separated by commas, white spaces, and new lines:

```
avec, et
mais pour
```

The following example shows how to create a **bts** index with a customized stopword list in a file:

```
CREATE INDEX books_bts ON books(book_data bts_lvarchar_ops)
USING bts(stopwords="file:/docs/stopwords.txt") IN bts_sbspace;
```

The following example shows how to create a **bts** index with a customized stopword list in a table column:

```
CREATE INDEX books_bts ON books(book_data bts_lvarchar_ops)
USING bts(stopwords="table:mytable.mycolumn") IN bts_sbspace;
```

Example 3: Create stopword lists for specific fields

The following example creates a stopword list of `am`, `be`, and `are` for all fields except the fields `author` and `title`, which have their own stopwords, and the field `edition`, which does not have any stopwords.

```
CREATE INDEX books_bts ON books(book_data bts_lvarchar_ops)
USING bts(stopwords=
    "(am,be,are,
      author:mrs,mr,ms,
      title:the,an,a,or,
      edition:)"
       )
IN bts_sbspace;
```

thesaurus index parameters

You can create a thesaurus so that basic text searches return synonyms as well as exact matches of specified words. A thesaurus is useful if your text data has multiple words for the same information.

thesaurus index parameters

" { **thesaurus="yes"** | **thesaurus_index** ="*thesaurus_index*" } "

Element	Description
thesaurus_index	The name of the **bts** index that is created on the thesaurus table.

Usage

People's names is an example of the type of data that can benefit from a thesaurus. Because people can have nicknames, multiple names for the same person might exist in the database. If you define a thesaurus for common nicknames, your basic text queries can return more accurate results. Synonyms are not used in a query if the query includes the following term modifiers: wildcard, fuzzy, proximity, or range query.

When you include a thesaurus in your **bts** index definition, basic text queries include all synonyms for specific search terms. For example, if you define mark, marc, marcus, and marco as synonyms, when you query for any one of these names the query is rewritten to include all of them:

```
'(mark OR marc OR marcus OR marco)'
```

To create a thesaurus:

1. Create the thesaurus table with a text column for the synonym data. You can use any of the data types that are supported by the **bts** index.
2. Add the synonym data to the thesaurus table. Each value for the synonym data column is a list of words that you want to be treated as synonyms. You can create synonyms for only single words. You cannot create synonyms for phrases.
3. Create a **bts** index on the thesaurus table. Include the thesaurus="yes" parameter.

When you create the **bts** index on the table that contains the text data, follow these rules:

- Specify the synonym data column as the column to index.
- Include the thesaurus_index="*thesaurus_index*" parameter, specifying the thesaurus index that you created.
- Set the query_default_operator index parameter to "OR" or omit the parameter.

You can dynamically update your thesaurus without rebuilding the basic text search index by updating the thesaurus table.

Example

Example

Suppose that you create a table called **mytbl** with the following statements:

```
CREATE TABLE mytbl(name char(30));
INSERT INTO mytbl(name) VALUES('mark');
```

```
INSERT INTO mytbl(name) VALUES('elizabeth');
INSERT INTO mytbl(name) VALUES('marco');
INSERT INTO mytbl(name) VALUES('beth');
```

You create a thesaurus table named **mythesaurus** and add synonym data to it:

```
CREATE TABLE mythesaurus(synonyms lvarchar);
INSERT INTO mythesaurus(synonyms)
      VALUES('elizabeth liz beth eliza leisal betty liza');
INSERT INTO mythesaurus(synonyms)
      VALUES('mark marc marcus marco');
```

You create a **bts** index on the thesaurus table:

```
CREATE INDEX mythesaurus_index
ON mythesaurus(synonyms bts_lvarchar_ops)
USING bts(thesaurus="yes");
```

You create a **bts** index that uses the thesaurus on the table **mytbl**:

```
CREATE INDEX name_index
ON mytbl(name bts_char_ops)
USING bts(thesaurus_index="mythesaurus_index");
```

Now when you search for the name `elizabeth`, the query returns both the exact match and the synonym `beth`:

```
SELECT * FROM mytbl WHERE bts_contains(name, 'elizabeth');

name
elizabeth
beth
2 row(s) retrieved.
```

When you search for both `marcus` or `liza`, the query returns four synonyms but no exact matches:

```
SELECT * FROM mytbl WHERE bts_contains(name, 'marcus or liza');

name
mark
marco
elizabeth
beth
4 row(s) retrieved.
```

xact_memory index parameter

You can set a limit on the amount of memory that is used by basic text search operations. Restricting memory usage is useful if you have a low memory configuration. By default, the memory limit is set by the value of the SHMTOTAL configuration parameter.

xact_memory index parameter

"**xact_memory**= { *memory_size* [{ **K | M | G** }] **| unlimited** }"

Element	Description
memory_size	A positive integer that represents the maximum amount of memory for **bts** index operations. The default unit is bytes. To specify a different multiple of bytes, include one of the following letters at the end of the number: • K = Kilobytes • M = Megabytes • G = Gigabytes For example, `5G` sets the maximum memory usage to 5 gigabytes.

Usage

If any **bts** index operation requires more memory than the value of the xact_memory index parameter, the operation fails.

If the xact_memory index parameter is set to unlimited or is not included in the index, the memory limit is set by the value of the SHMTOTAL configuration parameter.

Example

Example

For example, the following statement creates a **bts** index that limits the amount of memory for basic text search transactions to 5 GB:

```
CREATE INDEX books_bts ON books(book_data bts_lvarchar_ops)
USING bts(xact_memory=5G) IN bts_sbspace;
```

xact_ramdirectory index parameter

By default, you build a **bts** index in a temporary sbspace. You can build a **bts** index faster in RAM than in a temporary sbspace.

Include the xact_ramdirectory="yes" in the **bts** index to build the index in memory. However, when building the index in memory uses too much memory, the build is switched to the temporary sbspace. The maximum amount of memory that is allowed for an index build is approximately one third of the value of that is specified by the xact_memory index parameter or the SHMTOTAL configuration parameter, whichever is more restrictive.

Basic text search queries

You run basic text search queries with the bts_contains() search predicate.

You can run many types of basic text searches, such as word, phrase, Boolean, proximity, and fuzzy. You include the bts_contains() search predicate in the FROM clause of your query. Before you can run a search, you must create a **bts** index on the column you want to search.

Basic text search queries are not case-sensitive.

Searching on multiple columns

To run a basic text search query on multiple columns, you can create a composite **bts** index on those columns. If you include the query_default_field="*" index parameter, each column is indexed separately and you can run queries like the following query:

```
SELECT * FROM address WHERE bts_contains(fname, 'john AND city:nipigon');
```

Alternatively, you can create a different **bts** index on each column. However, you cannot use the SQL Boolean predicates AND, OR, and NOT between multiple bts_contains() search predicates in the same predicate clause. For example, the expression, `bts_contains(fname, 'john') AND bts_contains(lname, 'smith')` is not supported. To query on multiple bts indexes, use a UNION operator to join multiple SELECT statements that each include a different column in the bts_contains() search predicate.

Including the INDEX optimizer directive to force index scans

If you receive BTS22 errors from your queries, the optimizer might not be running the bts_contains() search predicate as an index scan. To force the optimizer to run the bts_contains() search predicate as an index scan, include the INDEX optimizer directive in your query. For example, the following query includes the INDEX optimizer directive for the **bts_idx** index on the **address** table:

```
SELECT INDEX(address bts_idx) * FROM address
        WHERE bts_contains(fname, 'john AND city:nipigon');
```

Basic Text Search query syntax

Use the bts_contains() search predicate to run basic text search queries.

You can also run a basic text search query with the HCL OneDB™ JSON $ifxtext query operator. Use the same syntax for the search criteria for both methods.

bts_contains() Search Predicate

"**bts_contains**(*column*, ' <Search criteria>' "

"[,**score # REAL**]"

")"

Search criteria

"[<Index field>: (explicit id)] *query_string*"

column

The column to be searched. It must be a single column for which a **bts** index is defined.

query_string

The search string. The search string includes the following elements:

Query term

Required. One or more words that you want to search for.

Query term modifiers

Optional. You can modify query terms to run wildcard, fuzzy, proximity, and range searches. You can boost the importance of a query term relative to other terms.

Boolean operators

Optional. You can include Boolean operators to combine query terms in logical combinations.

If an index has multiple fields because it is a structured or a composite index, you can include an index field name to modify the search string.

score # REAL

Optional argument that is used to pass a statement local variable (SLV) to the text search engine. The search engine uses this variable to record the document score it assigns to each row in the results. The score value is a REAL number between 0.0 and 100.0 inclusive that indicates the relevance of each document to the search criteria, compared to that of other indexed records. The higher the document score value, the more closely the document matches the criteria.

The following example shows a search for the word `standard` in the column **brands** in a table called **products**.

```
SELECT id FROM products
WHERE bts_contains(brands, 'standard');
```

You can use an SLV as a filtering mechanism and to sort the results by score. The following example returns documents that contain the word `standard` from the column **brands** in a table that is called **products** if the document score value is greater than 70. The results are ordered in descending order by score.

```
SELECT id FROM products
WHERE bts_contains(brands, 'standard', score # REAL)
AND score > 70.0;
ORDER BY score DESC;
```

Basic Text Search query terms

Query terms are words or phrases.

A word is a single word, such as `Hello`. A phrase is a group of words that are enclosed in double quotation marks, such as `"Hello World"`. Multiple words or phrases can be combined with Boolean operators to form complex queries.

This example searches for the word `Coastal`:

```
bts_contains(column, 'Coastal')
```

This example searches for the phrase `"Black and Orange"`:

```
bts_contains(column, ' "Black and Orange" ')
```

White space and punctuation characters are ignored. Terms within angle brackets (< >) are not interpreted as tagged HTML or XML text unless you are using XML index parameters. Letter case is not considered in query terms. Words are indexed in lowercase according to the DB_LOCALE environment variable setting. All three of the following search predicate examples search for the term `orange8` in unstructured text:

```
bts_contains(column, ' Orange8 ')
```

```
bts_contains(column, ' <oranGe8> ')
```

```
bts_contains(column, ' "<Orange8>" ')
```

Grouping words and phrases

You can group words and phrases in parentheses to form more complex queries by including Boolean operators. For example, to search for words `UNIX` or `Windows` and the phrase `operating system`, you can use this search predicate:

```
bts_contains(column, ' (UNIX OR Windows) AND "operating system" ')
```

This search returns results that must contain the phrase `operating system`, and either `the word UNIX` or the word `Windows`.

You can also group words and phrases in field data:

```
bts_contains(column, ' os:(UNIX AND "Windows XP") ')
```

In that case, the search results must contain the word `UNIX` and the phrase `Windows XP` in the `os` field.

Escaping special characters

You can use the special characters that are part of basic text search query syntax in searches by using the backslash (\) as an escape character before the special character.

The following characters are Basic Text Search special characters: + - && || ! ( ) { } [ ] ^ " ~ * ? : \

For example, to search for the phrase `(7+1)`, use the following search predicate:

```
bts_contains(column, ' \(7\+1\) ')
```

Basic text search index fields

The **bts** index indexes searchable data in *fields*. When you index unstructured text, each value is indexed in a default field called `contents`. You do not need to specify the default field in the bts_contains() search predicate. When you create an index that has multiple fields because it is a structured or a composite index, you might need to include a field name to modify the search string in the bts_contains() search predicate.

Index fields

" { [*JSONpath* .] | / *XMLpath* / | [*XMLnamespace* \:] } *fieldname* "

fieldname

The name of the field that is indexed.

JSONpath

If the json_path_processing index parameter is enabled, you can include the path before the field name. Separate each part of the path with a period.

XMLpath

If the xml_path_processing index parameter is enabled, you can include the path before the field name. Separate each part of the path with a forward slash.

XMLnamespace

If the include_namespaces index parameter is enabled, you can include an XML namespace before the field name. Escape the colon in the namespace with a back slash.

If you create a composite index on multiple columns, by default the text from the indexed columns is concatenated into one string and indexed in the `contents` field. To index the text in each column included in the index under a field of the same name, include the query_default_field="*" index parameter in the index definition. When you query on a composite index that has multiple fields, you must specify the field name in the bts_contains() search predicate.

Searches on structured JSON or XML indexes

When you index structured text by setting XML or JSON index parameters, the names for the XML tags or JSON field names are indexed in separate fields and you must specify those fields in the bts_contains() search predicate.

If you specify a list of XML tags or JSON field names to be indexed with the xmltags or json_names index parameter, the default field is the first field in the field list. You must specify the field name for any other field in the bts_contains() search predicate. However, you can override the default field by setting the query_default_field index parameter to a specific field name to use as the default field.

If you enable the **all_xmltags** or all_json_names index parameter, there is no default field. You must specify each field name in the bts_contains() search predicate.

Searches on structured XML indexes

When you index structured text by setting XML index parameters, the names for the XML tags or paths are indexed in separate fields and you must specify those fields in the bts_contains() search predicate. If you specify a list of XML tags to be indexed with the xmltags index parameter, the default field is the first tag or path in the field list. You must specify the field name for any other field in the bts_contains() search predicate. If you enable the **all_xmltags** index parameter, there is no default field. You must specify each field name in the bts_contains() search predicate.

To search text within a field, specify the field name followed by a colon (:) and the query term in the format *fieldname*:*string*.

Example

Examples: JSON or BSON documents

For these examples, the following JSON document is indexed as field name-value pairs with paths by enabling the all_json_names and json_path_processing index parameters:

```
{ “person” : {
     “givenname” : “Jim”
}
```

For example, to search the `given name` field, you can use either of the following search predicates:

```
bts_contains(column, ' givenname:Jim ')
```

```
bts_contains(column, ' givenname:"Jim" ')
```

To search for a field that includes a path, include a period between the field name elements. For example, to search the `person:given name` field, you can use the following search predicate:

```
bts_contains(column, ' person.givenname:"Jim" ')
```

Example

Examples: XML documents

For example, if the XML data is indexed in a field that is called `fruit`, you can use the following search predicates:

```
bts_contains(column, ' fruit:Orange ')
```

```
bts_contains(column, ' fruit:"Orange Juice" ')
```

If the XML data is indexed in a field that contains the path `/fruit/citrus`, you can use the following search predicate:

```
bts_contains(column, ' /fruit/citrus:"Orange Juice" ')
```

If you enable the include_namespaces index parameter, you must escape the colon (:) in namespaces with a backslash (\). For example, if you are using the `fruit` namespace:

```
bts_contains(column, ' fruit\:citrus:Orange ')
```

Basic Text Search query term modifiers

You can modify query terms to perform more complex searches.

If you are searching fielded data, you can use query term modifiers only on the query terms, not on the field names.

Wildcard searches

You can use wildcards in basic text search queries on single terms. You cannot use wildcards in searches on phrases.

To perform a single-character wildcard search, use a question mark (?) in the search term. The single-character wildcard search looks for terms that match with the single character replaced. For example, to search for the terms `text` and `test`, use `te?t` in the search predicate:

```
bts_contains(column, 'te?t')
```

You can use a single wildcard character (?) as the first character of the search term.

Multiple-character wildcard searches

Multiple-character wildcard searches look for zero or more characters.

To perform a multiple-character wildcard search, use an asterisk (*) in the search term. For example, to search for `geo`, `geography`, and `geology`, use `geo*` in the search predicate:

```
bts_contains(column, 'geo*')
```

The multiple-character wildcard search can also be in the middle of a term. For example, the search term `c*r` will match `contour`, `crater`, `color`, and any other words that start with the letter c and end with the letter r:

```
bts_contains(column, 'c*r')
```

You cannot use a multiple wildcard character (*) as the first character of the search term.

If the number of indexed tokens that match your wildcard query exceed 1024, you receive the following error:

```
(BTSB0) - bts clucene error: Too Many Clauses
```

To solve this problem, you can make the query more restrictive or you can recreate the **bts** index with the max_clause_count index parameter set to a number greater than 1024.

Fuzzy searches

A fuzzy search searches for text that matches a term closely instead of exactly. Fuzzy searches help you find relevant results even when the search terms are misspelled.

To perform a fuzzy search, append a tilde (~) at the end of the search term. For example the search term `bank~` will return rows that contain `tank`, `benk` or `banks`.

```
bts_contains(column, 'bank~')
```

You can use an optional parameter after the tilde in a fuzzy search to specify the degree of similarity. The value can be between 0 and 1, with a value closer to 1 requiring the highest degree of similarity. The default degree of similarity is 0.5, which means that words with a degree of similarity greater than 0.5 are included in the search.

The degree of similarity between a search term and a word in the index is determined by using the following formula:

```
similarity = 1 - (edit_distance / min ( len(term), len(word) ) )
```

The edit distance between the search term and the indexed word is calculated by using the Levenshtein Distance, or Edit Distance algorithm. The min() function returns the minimum of the two values of the len() functions, which return the length

of the search term and the indexed word. The following table shows the values used to calculate similarity and the resulting similarity between the search term "tone" and various indexed words.

Table 21. Sample set of comparisons

Term	Length of term	Word	Length of word	Edit distance	Similarity
tone	4	tone	4	0	1.00
tone	4	ton	3	1	0.67
tone	4	tune	4	1	0.75
tone	4	tones	4	1	0.75
tone	4	once	4	2	0.50
tone	4	tan	3	2	0.33
tone	4	two	3	3	0.00
tone	4	terrible	8	6	-0.50
tone	4	fundamental	11	9	-1.25

For example, the following query searches for words with the default degree of similarity of greater than 0.50 to the search term `tone`:

```
bts_contains(text, 'tone~')
```

This query returns rows that contain these words: `tone`, `ton`, `tune`, and `tones`. Rows that contain the word `once`are not included because the degree of similarity for `once` is exactly 0.50, not greater than 0.50. The following query would include the rows that contain the word `once`:

```
bts_contains(text, 'tone~0.49')
```

Tip: Test the behavior of specifying the degree of similarity with your data before you rely on it in your application.

If the number of indexed tokens that match your fuzzy query exceed 1024, you receive the following error:

```
(BTSB0) - bts clucene error: Too Many Clauses
```

To solve this problem, you can make the query more restrictive or you can recreate the **bts** index with the max_clause_count index parameter set to a number greater than 1024.

Proximity searches

You can specify the number of nonsearch words that can occur between search terms in a proximity search.

To perform a proximity search, enclose the search terms within double quotation marks and append a tilde (~) followed by the number of nonsearch words allowed. For example, to search for the terms `curb` and `lake` within 8 words of each other within a document, use the following search predicate:

```
bts_contains(column, ' "curb lake"~8 ')
```

Range searches

With a range search, you match terms that are between the lower and upper bounds specified by the query. Range searches can be inclusive or exclusive of the upper and lower bounds. Sorting is in lexicographical order (also known as dictionary order or alphabetic order).

Lexicographical order does not give the expected results to numeric data unless all numbers have the same number of digits. If necessary, add zeros to the beginning of numbers to provide the necessary number of digits.

Range searches use the keyword TO to separate search terms. By default, the word "to" is a stopword and is not an indexed term. If you are using a stopword list that does not include the word "to" or you are not using a stopword list, omit the word TO from the range query.

Inclusive range searches

Use brackets ([ ]) in the search predicate to specify an inclusive search. The syntax is [*searchterm1* TO *searchterm2*].

The following search predicate finds all terms between `apple` and `orange`, including the terms `apple` and `orange`:

```
bts_contains(column, ' [apple TO orange] ')
```

This example finds all terms between `20063105` and `20072401`, including `20063105` and `20072401`:

```
bts_contains(column, ' [20063105 TO 20072401] ')
```

Exclusive range searches

Use braces ({ }) in the search predicate to specify an exclusive search. The syntax is {*searchterm1* TO *searchterm2*}.

The following search predicate finds all terms between `Beethoven` and `Mozart`, excluding the terms `Beethoven` and `Mozart`:

```
bts_contains(column, ' {Beethoven TO Mozart} ')
```

This example finds all terms between `65` and `89`, excluding `65` and `89`:

```
bts_contains(column, ' {65 TO 89} ')
```

Boost a term

Boosting a term assigns more relevance to a word or phrase.

By default, all terms have equal value when the relevance score of a matching document is computed. Boosting a term raises the score of a document that contains it above the score of documents that do not. The search results are the same, but when sorted in descending order by score, documents containing the boosted term appear higher in the results.

To boost a term, use the caret symbol (^) followed by a number for the boost factor after the term that you want to appear more relevant. By default the boost factor is 1. It must be a positive number, but it can be less than one: for example .3 or .5.

For example, if your search terms are `Windows` and `UNIX` as in the search predicate `bts_contains(column, ' Windows UNIX ')`, you can boost the term `Windows` by a factor of 4:

```
bts_contains(column, ' Windows^4 UNIX ')
```

This example boosts the phrase `road bike` over the phrase `mountain bike` by a factor of 2:

```
bts_contains(column, ' "road bike"^2 "mountain bike"  ')
```

You can also boost more than one term in a query. This example would return rows with the term `lake` before documents with the term `land`, before documents with the term `air`.

```
bts_contains(column, ' lake^20  land^10 air  ')
```

Tip: Test the behavior of boosting a term with your data before you rely on it in your application.

Boolean operators

Boolean operators combine terms in logical combinations. You can use the operators AND, OR, and NOT, or their equivalent special characters, in the bts_contains() search predicate.

By default, the OR operator is assumed if you do not supply a Boolean operator between two terms. However, you change the default operator to AND by setting the **query_default_operator** to AND when you create a **bts** index. For more information, see bts access method syntax on page 122.

The Boolean operators are not case-sensitive.

AND operator

The AND operator matches documents where both terms exist anywhere in the text of a single document.

You can also use two adjacent ampersands (&&) instead of AND.

If the **query_default_operator** index parameter is set to AND, the AND operator is assumed if you do not specify a Boolean operator between two terms.

The following search predicates search for documents that contain both the word `UNIX` and the phrase `operating system`:

```
bts_contains(column, ' UNIX AND "operating system"  ')
```

```
bts_contains(column, ' UNIX && "operating system"  ')
```

The following search predicates search XML data for documents that contain both the word `travel` in the `book` field and the word `stewart` in the `author` field:

```
bts_contains(column, ' book:travel AND author:stewart  ')
```

```
bts_contains(column, ' book:travel && author:stewart  ')
```

The following search predicate searches for documents that contain both the word `travel` in the `book` field and the phrase `john stewart` in the `author` field:

```
bts_contains(column, ' book:travel AND author:"john stewart" ')
```

OR operator

The OR Boolean operator is the default conjunction operator. If no Boolean operator appears between two terms, the OR operator is assumed, unless the **query_default_operator** index parameter is set to AND. In that case, you must specify the OR operator, or use two adjacent vertical bars (||) to represent the OR operator.

The following search predicates find documents that contain either the term `UNIX` or the term `Windows`:

```
bts_contains(column, ' UNIX Windows ')
```

```
bts_contains(column, ' UNIX OR Windows ')
```

```
bts_contains(column, ' UNIX || Windows ')
```

NOT operator

Use the NOT Boolean operator in combination with the AND operator (or its equivalent symbols) when you want to search for documents that do not contain a specified term or phrase.

The NOT operator can also be denoted with an exclamation point (!) or with a minus sign (-).

The following search predicates find documents that contain the term `UNIX`, but not the term `Windows`:

```
bts_contains(column, ' UNIX AND NOT Windows ')
```

```
bts_contains(column, ' UNIX AND !Windows ')
```

```
bts_contains(column, ' +UNIX -Windows ')
```

The minus sign (-) can be used with the plus sign (+), but not with the AND operator.

Basic text search JSON index parameters

You can include JSON index parameters when you create a **bts** index to control how JSON and BSON columns are indexed.

By default, all field names and values are indexed as unstructured text in the `contents` field. Use JSON index parameters to control the following aspects of the **bts** index:

- Whether to index the documents as field name-value pairs so that you can search for text by field. Enable the all_json_names index parameter to index all field names. Set the json_names index parameters to index specific field names. You have the following choices to further refine how field name-value pairs are indexed:
    - Whether to include JSON or BSON object paths in field name-value pairs so that you can search based on the field hierarchy in the document. Enable the json_path_processing index parameter to index paths.
    - Whether to index the position of values in arrays so that you can search specific positions in arrays. Enable the json_array_processing index parameter to index the position of arrays.
    - Whether to index as both field name-value pairs and unstructured text so that you have the flexibility to search a specific field or all fields. Enable the include_contents index parameter to include an unstructured index of field names and values.

- Whether an unstructured index contains only values and no field names so that you do not receive field names in search results. Enable the only_json_values index parameter to limit the unstructured index to values.
- Whether to ignore format errors for JSON or BSON documents. Enable the ignore_json_format_errors index parameter to ignore incorrectly formatted documents.

Requirements and restrictions

The JSON or BSON documents must be in a UTF-8 locale.

Any XML values in a JSON or BSON document are indexed as unstructured text.

The following parts of JSON or BSON documents are indexed by a **bts** index:

- JSON string values, or the corresponding BSON element code 0x2.
- JSON number values, which are converted to string representations: 4-byte integers, 8-byte integers, and 8-byte floating points, or the corresponding BSON element codes: \x01, \x09, \x10, \x11, and \x12.
- JSON TimeStamp and Coordinated Universal Time Datetime values, which are converted to string representations

The following parts of JSON or BSON documents are not indexed:

- JSON Boolean true, Boolean false, and null values
- The BSON element codes: 0x05, 0x06, 0x07, 0x08, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0xFF, and 0x7F.
- Any name-value pair that has a zero-length field name
- Fields that contain numbers

You cannot create a composite index on a JSON or BSON column.

Example document

The examples for indexing JSON and BSON documents are based on the following JSON document, which is assumed to be in the **docs** column of the **json_tab** table:

```
{ "person" : {
     "givenname" : "Jim",
     "surname" : "Flynn",
     "age" : 29,
     "cars" : [ "dodge", "olds" ],
     "parents":[
        { "givenname" : "Slim",
          "surname" : "Flynn" },
        { "givenname" : "Lynn",
          "surname" : "Kim" }
     ]
}
```

The **bts** index on a JSON or BSON document is based on a tree representation of the document. You need to understand the tree representation if you include paths or array positions in the field name-value pairs of a structured index. The example JSON document has the following tree representation:

```
"person".
         "givenname" : "Jim"
         "surname” : "Flynn"
         "age" : "29",
         "cars".
                "0" : "dodge"
                "1" : "olds"
         "parents".
                   "0".
                       "givenname" : "Slim"
                       "surname" : "Flynn"
                   "1".
                       "givenname" : "Lynn"
                       "surname" : "Kim"
```

JSON index parameters syntax

You can use JSON index parameters to index the contents of JSON and BSON columns as structured or unstructured text, or both.

Include JSON index parameters in the **bts** index definition when you create the **bts** index. See bts access method syntax on page 122. You can also create a **bts** index on a BSON column by running the HCL OneDB™ JSON createTextIndex command. Both methods requires the same syntax for JSON and other **bts** index parameters.

You can index JSON or BSON documents as structured or unstructured text.

JSON index parameters for structured text

“ { <The json_names index parameter> (explicit id) | **all_json_names**= { **"yes"** | **"no"** } (explicit id) } ”

“ [, **include_contents** = { **"yes"** | **"no"** } (explicit id) [, **only_json_values** = { **"yes"** | **"no"** } (explicit id)]] ”

“ [, **json_path_processing** = { **"yes"** | **"no"** } (explicit id)] ”

“ [, **json_array_processing**= { **"yes"** | **"no"** } (explicit id)] ”

“ [, **ignore_json_format_errors**= { **"yes"** | **"no"** } (explicit id)] ”

JSON index parameters for unstructured text

“ [**only_json_values**= { **"yes"** | **"no"** } (explicit id)] ”

“ [, **json_array_processing**= { **"yes"** | **"no"** } (explicit id)] ”

“ [, **ignore_json_format_errors**= { **"yes"** | **"no"** } (explicit id)] ”

Usage

If you do not include any JSON index parameters when you create a **bts** index on a JSON or BSON column, both the field names and the values are indexed together as unstructured text.

Include a comma between parameters.

Example

Example

The following statement creates a **bts** index without JSON index parameters on the example JSON **docs** column:

```
CREATE INDEX bts_idx
         ON json_tab (docs bts_json_ops)
         USING bts;
```

The resulting index contains the following unstructured text that is based on the tree representation of the document in the `contents` field:

```
contents: person givenname jim surname flynn age 29 cars dodge olds parents
givenname slim surname flynn givenname lynn surname kim
```

all_json_names index parameter

Enable the all_json_names index parameter to index JSON or BSON documents as field name-value pairs instead of as unstructured text.

All the field names in the documents in the column are indexed as fields in the **bts** index. When you query on the JSON or BSON column, you must specify the field name to search in the bts_contains() search predicate.

You can include the json_path_processing and json_array_processing index parameters to add the paths and array positions to the field names.

To view the fields that you indexed, run the bts_index_fields() function.

Example

Example: Index all field name-value pairs

The following statement creates a **bts** index with the all_json_names index parameter enabled on the example JSON **docs** column:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(all_json_names="yes");
```

The resulting index contains the following field name-value pairs:

```
givenname: jim
surname: flynn
givenname: slim
surname: flynn
```

```
age: 29
cars: dodge
cars: olds
givenname: lynn
surname: kim
```

You can specify the following fields in the search predicate: **givenname**, **surname**, **age**, and **cars**.

ignore_json_format_errors index parameter

Enable the ignore_json_format_errors index parameter to skip inserting incorrectly formatted JSON or BSON documents and continue processing the SQL statement. By default, if you attempt to insert a JSON or BSON document that contains a format error, such as a missing brace or bracket, the entire SQL statement fails.

When you create a **bts** index or have an existing **bts** index and you insert JSON or BSON documents, the database server checks the formatting of the documents. When you enable the ignore_json_format_errors index parameter, incorrectly formatted documents are not inserted, but the rest of the statement continues processing. Any skipped documents result in messages in the online message log.

Example

Example

The following statement creates a **bts** index with the ignore_json_format_errors index parameter enabled on the example JSON **docs** column:

```
create index bts_idx
          on json_tab (docs bts_json_ops)
       using bts(ignore_json_format_errors="yes");
```

include_contents index parameter

Enable the include_contents index parameter to index JSON or BSON documents as unstructured text as well as indexing the documents as field name-value pairs.

You can enable the include_contents index parameter if the all_json_names parameter is enabled or the json_names parameter is specified.

By default, both field names and values are indexed in the `contents` field. If you enable the only_json_values index parameter, only the values are indexed in the `contents` field.

Example

Example: Index all fields as field name-value pairs and unstructured text

The following statement creates a **bts** index with the all_json_names and include_contents index parameters enabled on the example JSON **docs** column:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(
```

```
                all_json_names="yes",
                include_contents="yes");
```

The resulting index contains the following 9 field name-value pairs, and the field names and values as unstructured text in the `contents` field:

```
givenname: jim
givenname: slim
givenname: lynn
age: 29
cars: dodge
cars: olds
surname: flynn
surname: flynn
surname: kim
contents: person givenname jim surname flynn age 29 cars dodge olds
parents givenname slim surname flynn givenname lynn surname kim
```

Example

Example: Index specified field name-value pairs, paths, and values as unstructured text

The following statement creates a **bts** index with the json_names, json_path_processing, only_json_values, and include_contents index parameters enabled on the example JSON **docs** column:

```
create index bts_idx
          on json_tab (docs bts_json_ops)
       using bts(
                json_names="(person.givenname,parents.surname)",
                json_path_processing="yes",
                include_contents="yes",
                only_json_values="yes");
```

The resulting index contains four fields: three field name-value pairs with paths and the unstructured text in the `contents` field:

```
person.givenname: jim
parents.surname: flynn
parents.surname: kim
contents: jim flynn 29 dodge olds slim flynn lynn kim
```

json_array_processing index parameter

Enable the json_array_processing index parameter to index the array positions of values in JSON or BSON documents as field names.

Array positions are numbers, starting with 0, which represent the position of the value in the array. For example, the array `"cars" : [ "dodge", "olds" ]` has two positions:

```
"cars".
        "0" : "dodge"
        "1" : "olds"
```

In this example, the field name for `dodge` is `0` and the field name for `olds` is `1`. Field names that are only numbers cannot be queried, and are therefore not indexed. If you index field name-value pairs and array positions, but not paths, then field name-value pairs in arrays are not indexed, because the field names are numbers.

Indexing the array positions is most useful when you also index field name-value pairs and paths. Array positions in a field name that includes a path are indexed because the field name contains more than just a number. For example, the field names with paths from the example array are `cars.0` and `cars.1`.

When array and path processing are both enabled, the paths specified in the json_names index parameter must include array positions.

Example

Example: Index array positions

The following statement creates a **bts** index with the json_array_processing index parameter enabled on the example JSON **docs** column:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(
                 json_array_processing="yes");
```

The resulting index indexes the following unstructured text that contains the values and the array positions in the document in the `contents` field:

```
contents: person givenname jim surname flynn age 29 cars 0 dodge 1 olds
parents 0 givenname slim surname flynn 1 givenname lynn surname kim
```

In this example, indexing the array positions does not provide meaningful index entries because the position numbers are not differentiated from other values. If you query for the number 1, you do not know if the number is a value or an array position. Array positions are meaningful only in the context of field names and paths.

Example

Example: Index all field name-value pairs, paths, and array positions

The following statement creates a **bts** index with the all_json_names, json_path_processing, and json_array_processing index parameters enabled on the example JSON **docs** column:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(
                 all_json_names="yes",
                 json_path_processing="yes",
                 json_array_processing="yes");
```

The resulting index contains the following field name-value pairs that contain paths and array positions:

```
person.givenname: jim
person.surname: flynn
person.age: 29
person.cars.0: dodge
```

```
person.cars.1: olds
person.parents.0.givenname: slim
person.parents.0.surname: flynn
person.parents.1.givenname: lynn
person.parents.1.surname: kim
```

Example

Example: Index specified field name-value pairs, paths, and array positions

The following statement creates a **bts** index with the json_names, json_path_processing, and json_array_processing index parameters enabled on the example JSON **docs** column:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(
                 json_names="(person.givenname,parents.1.surname)",
                 json_path_processing="yes",
                 json_array_processing="yes");
```

The array position is required. If you specify `parents.surname` instead of `parents.1.surname`, this example results in an error.

The resulting index contains the following field name-value pairs that contain paths and array positions:

```
person.givenname: jim
parents.1.surname: kim
```

Example

Example: Index all field name-value pairs and array positions

The following statement creates a **bts** index with the all_json_names and json_array_processing index parameters enabled on the example JSON **docs** column:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(
                 all_json_names="yes",
                 json_array_processing="yes");
```

The resulting index contains the following field name-value pairs and array positions:

```
givenname: jim
givenname: slim
givenname: lynn
age: 29
surname: flynn
surname: flynn
surname: kim
```

The following field name-value pairs are not indexed because the field names are numbers:

```
0: dodge
1: olds
```

json_names index parameter

Enable the indexing of specific field name-value pairs in JSON or BSON documents with the json_names index parameter.

The input for the field names for the **json_names** index parameter can be a comma-separated list of names, an external file, or a table column.

The json_names index parameter

" `json_names="` { `(` *field* `)` | `file:`*directory/filename* | `table:`*table.column* } `"` "

Table 22. Elements for the json_names index parameter

Element	Description
column	The column that contains the field names to index. Separate field names by commas, white spaces, or new-line characters.
directory	The location of the file that contains field names to index.
field	The field name to index.
filename	The name of the file that contains field names to index. Must be readable by the user who creates the index. Separate field names by commas, white spaces, or new-line characters.
table	The name of the table with the column that contains the field names to index. Must be readable by the user who creates the index.

The field names that you specify are indexed as fields in the **bts** index. The values in the fields can be searched. When you query on the JSON or BSON column, you must specify the field name to search in the bts_contains() search predicate. In searches, the default field is the first tag or path in the field list. The **bts** index does not check whether the fields exist in the column, which means that you can specify fields that you will add to the column after you create the index.

If you enable the json_path_processing index parameter, the field name can include relative or full paths. If you enable the json_array_processing index parameter, the field name can include array positions.

If you want to add new field names to the index, you must drop the index, update the field name list, and then re-create the index.

To view the fields that you indexed, run the bts_index_fields() function.

Example

Example: Index one field name-value pair

The following statement creates a **bts** index with the json_names index parameter set to a single field name on the example JSON **docs** column:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(json_names="surname");
```

The resulting index contains the following field name-value pairs:

```
surname: flynn
surname: flynn
surname: kim
```

You must specify the **surname** field in the search predicate.

Example

Example: Index field name-value pairs from a file

The following statement creates a **bts** index with the json_names index parameter set to a file that is named `jsonfield.txt`:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(json_names="file:/jsonfield.txt");
```

Example

Example: Index field name-value pairs from a column

The following statement creates a **bts** index with the json_names index parameter set to a column that is named **jsonnames** in the **json_ref** table:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(json_names="table:json_ref.jsonnames");
```

json_path_processing index parameter

Enable the json_path_processing index parameter to include paths as part of the field names in field-value pairs from JSON or BSON documents.

You can enable the json_path_processing index parameter if you enable the indexing of field name-value pairs with either one of the following index parameters:

- The json_names index parameter: In the list of fields to index, you can specify relative paths or full paths. For example, if the full path is `person.parents.surname`, you can specify the relative path `parents.surname`.
- The all_json_names index parameter: Full paths are indexed for all fields.

If you add the json_array_processing index parameter, the paths include array positions, for example: `person.cars.0`.

Example

Example: Index all field name-value pairs and paths

The following statement creates a **bts** index with the all_json_names and json_path_processing index parameters enabled on the example JSON **docs** column:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(
                 all_json_names="yes",
                 json_path_processing="yes");
```

The resulting index contains the following field name-value pairs that include paths:

```
person.givenname: jim
person.surname: flynn
person.age: 29
person.cars: dodge
person.cars: car
person.parents.givenname: slim
person.parents.surname: flynn
person.parents.givenname: lynn
person.parents.surname: kim
```

Example

Example: Index specified field name-value pairs and paths

The following statement creates a **bts** index with the json_names and json_path_processing index parameters on the example JSON **docs** column:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(
                 json_names="(parents.surname,
                              person.givenname)",
                 json_path_processing="yes");
```

The `parents.surname` path is a relative path instead of the full path, `person.parents.surname`.

The resulting index contains the following field name-value pairs that include paths:

```
person.givenname: jim
parents.surname: flynn
parents.surname: kim
```

only_json_values index parameter

Enable the only_json_values index parameter to index only the values in the JSON or BSON documents as unstructured text. The field names are not indexed.

You can index both field name-value pairs and values, but not field names, as unstructured text. The json_names or all_json_names index parameter enables the indexing of field name-value pairs. The include_contents index parameter

enables the indexing of field names and values as unstructured text. Add the only_json_values index parameter to modify the behavior of the include_contents index parameter to omit field names from the `contents` field.

Example

Example: Index values as unstructured text

The following statement creates a **bts** index with the only_json_values index parameter enabled on the example JSON **docs** column:

```
create index bts_idx
          on json_tab (docs bts_json_ops)
       using bts(only_json_values="yes");
```

The resulting index indexes the following unstructured text that contains only the values in the document in the `contents` field:

```
contents: jim flynn 29 dodge olds slim flynn lynn
```

Example

Example: Index all field name-value pairs and values as unstructured text

The following statement creates a **bts** index with the all_json_names, only_json_values, and include_contents index parameters enabled on the example JSON **docs** column:

```
create index bts_idx
          on json_tab (docs bts_json_ops)
       using bts(
                 all_json_names="yes",
                 include_contents="yes",
                 only_json_values="yes");
```

The resulting index contains the following 9 field name-value pairs and the values as unstructured text in the `contents` field:

```
givenname: jim
givenname: slim
givenname: lynn
age: 29
cars: dodge
cars: olds
surname: flynn
surname: flynn
surname: kim
contents: jim flynn 29 dodge olds slim flynn lynn kim
```

Basic Text Search XML index parameters

This chapter describes the XML index parameters for basic text search and provides detailed examples about each parameter's usage.

XML index parameters syntax

You can use XML index parameters to index XML tag and attribute values in separate fields either by tag name, attribute name, or by path.

When you do not use XML index parameters, XML documents are indexed as unstructured text. The XML tags, attributes, and values are included in searches and are indexed together in the `contents` field.

Any JSON or BSON documents in an XML document are indexed as unstructured text.

Include XML index parameters in the **bts** index definition when you create the **bts** index. See bts access method syntax on page 122.

XML Index Parameters

" { **xmltags="** { (*field*) | **file:***directory/filename* | **table:***table.column* } " (explicit id) | { | **all_xmltags=** { **"no"** | [**"yes"**] } (explicit id) | **all_xmlattrs=** { **"no"** | [**"yes"**] } (explicit id) } } "

" [{ | **include_contents=** { **"no"** | [**"yes"**] } (explicit id) | **xmlpath_processing=** { **"no"** | [**"yes"**] } (explicit id) | **strip_xmltags=** { **"no"** | [**"yes"**] } (explicit id) | **include_namespaces=** { **"no"** | [**"yes"**] } (explicit id) | **include_subtag_text=** { **"no"** | [**"yes"**] } (explicit id) }] "

Table 23. Options for XML index parameters

Element	Description
column	The column that contains tags to index.
directory	The location of the file that contains tags to index.
field	The XML tag or path to index. The field values can be full or relative XML paths if used with the xmlpath_processing parameter.
filename	The name of the file that contains tags to index.
table	The name of the table that contains the column with tags to index.

Example

Example

For example, you have the following XML fragment:

```
<skipper>Captain Black</skipper>
```

You can create a **bts** index for searching the text within the `<skipper>` `</skipper>` tags:

```
CREATE INDEX boats_bts ON boats(xml_data bts_lvarchar_ops)
USING bts(xmltags="(skipper)") IN bts_sbspace;
```

To search for a skipper's name that contains the word "Black," use the **bts** search predicate:

```
bts_contains(xml_data, 'skipper:black')
```

The xmltags index parameter

Use the **xmltags** parameter to specify which XML tags or XML paths are searchable in a column.

The XML tags or paths that you specify become the field names in the **bts** index. The text values within fields can be searched. In searches, the default field is the first tag or path in the field list. The Basic Text Search module does not check if the tags exist in the column, which means that you can specify fields for tags that you will add to the column after you have created the index.

The input for the field names for the **xmltags** parameter can be one of three forms:

- inline comma-separated values
- an external file
- a table column

Input as inline comma-separated field names

Inline comma-separated field names are useful when you have only a few fields to index. For example, `xmltags="(field1,field2,field3)"` where *fieldn* specifies the tag or path to index.

If the **xmltags** parameter is enabled, you can specify paths for the **xmltags** values. For example

```
xmltags="(/text/book/title,/text/book/author,/text/book/date)"
```

XML tags are case-sensitive. When you use the inline comma-separated field names for input, the field names are transformed to lowercase characters. If the field names are uppercase or mixed case, use an external file or a table column for input instead.

Input from a file or a table column

Input from an external file has the format: `xmltags="file:/directory/filename"`

Input from a table column has the format: `xmltags="table:table.column"`

The file or table that contains the field names must be readable by the user creating the index. The file or table is read only when the index is created. If you want to add new field names to the index, you must drop and re-create the index. The field names in the file or table column can be separated by commas, white spaces, newlines, or a combination.

Following is an example of how field names can appear in the file or the table column:

```
title, author
date ISBN
```

If the **xmlpath_processing** parameter is enabled, you can specify paths or combination of paths and individual field names in the file or the table column:

```
/text/book/title
author
```

For information about using XML paths, see The xmlpath_processing index parameter on page 160.

If you want to index all the XML tags in a column, see The all_xmltags index parameter on page 158.

To view the fields that you have indexed, use the bts_index_fields() function. See bts_index_fields() function on page 181.

Example: Index specific XML tags

You can use the **xmltags** parameter to index-specific fields so that you can restrict your searches by XML tag names.

Given the table:

```
EXECUTE PROCEDURE IFX_ALLOW_NEWLINE('t');

 CREATE TABLE boats(docid integer, xml_data lvarchar(4096));
 INSERT INTO boats values(1, '
  <boat>
   <skipper>Captain Jack</skipper>
   <boatname>Black Pearl</boatname>
  </boat> ');
 INSERT INTO boats values(2, '
  <boat>
   <skipper>Captain Black</skipper>
   <boatname>The Queen Anne's Revenge</boatname>
   </boat> ');
```

To create a **bts** index for the `skipper` and `boatname` tags:

```
CREATE INDEX boats_bts ON boats(xml_data bts_lvarchar_ops)
USING bts(xmltags="(skipper,boatname)") IN bts_sbspace;
```

The index will contain the following fields:

For the row where docid = 1, the fields are:

```
skipper:Captain Jack
boatname:Black Pearl
```

For the row where docid = 2, the fields are:

```
skipper:Captain Black
boatname:The Queen Anne's Revenge
```

To search for the skipper with the name " Black", the SELECT statement is:

```
SELECT xml_data FROM boats WHERE bts_contains(xml_data, 'skipper:black');
```

The search will return docid 2 because the `skipper` field for that row contains the word "black." For docid = 1, the `boatname` field also contains the word "black," but it is not returned because the search was only for the `skipper` field.

The all_xmltags index parameter

Use the **all_xmltags** parameter to enable searches on all the XML tags or paths in a column.

All the XML tags are indexed as fields in the **bts** index. If you use the **xmlpath_processing** parameter, full paths are indexed. The text value within fields can be searched. The attributes of XML tags are not indexed in a field unless you use the **all_xmlattrs** index parameter.

For information about using paths, see The xmlpath_processing index parameter on page 160.

If you want to index only specific tags in a column, use the **xmltags** parameter. See The xmltags index parameter on page 156.

To view the fields that you have indexed, use the bts_index_fields() function. See bts_index_fields() function on page 181.

Example: Index all XML tags

You can use the all_xmltags parameter to index all of the tags in a column.

Given the XML fragment:

```
<book>
 <title>Graph Theory</title>
 <author>Stewart</author>
 <date edition="second">January 14, 2006</date>
</book>
```

To create an index for all the XML tags, use the SQL statement:

```
CREATE INDEX book_bts ON books(xml_data bts_lvarchar_ops)
USING bts(all_xmltags="yes") IN bts_sbspace;
```

The index will contain three fields that can be searched:

```
title:graph theory
author:stewart
date:january 14, 2006
```

The top level `<book></book>` tags are not indexed because they do not contain text values. The `edition` attribute is also not indexed.

If you enable path processing with the **xmlpath_processing** parameter, you can index the full paths:

```
CREATE INDEX book_bts ON books(xml_data bts_lvarchar_ops)
USING bts(all_xmltags="yes",xmlpath_processing=”yes”) IN bts_sbspace;
```

The index will contain three fields with full paths that can be searched:

```
/book/title:graph theory
/book/author:stewart
/book/date:january 14, 2006
```

The all_xmlattrs index parameter

Use the **all_xmlattrs** parameter to search on XML attributes in a document repository stored in a column of a table. This parameter enables searches on all attributes that are contained in the XML tags or paths in a column that contains an XML document.

Specify an attribute using the syntax `@attrname`, where `attrname` is the name of the attribute.

All the XML attributes are indexed as fields in the **bts** index. If you use the **xmlpath_processing** parameter, full paths are indexed. The text value within fields can be searched. The tags of XML tags are not indexed in a field unless you use the **all_xmltags** index parameter.

To view the fields that you have indexed, use the bts_index_fields() function. See bts_index_fields() function on page 181.

Examples: Index XML attributes

These examples are based on the following three rows of data:

```
<boat><name reg="hmc">titanic</name></boat>

<airplane callsign="qofz">kittyhawk</airplane>

<boat><name reg="CAN">Spirit of Canada</name></boat>
```

Example 1: Compare all_xmltags and all_xmlattrs

The following CREATE INDEX statement uses the **all_xmltags** parameter:

```
CREATE INDEX bts_idx ON bts_100_tab(col2 bts_nvarchar_ops)
       USING bts(all_xmltags="yes") IN bts_sbspace1 ;
```

The index has these fields representing the type of tag:

```
airplane
name
```

By contrast, the following CREATE INDEX statement uses the **all_xmlattrs** parameter instead of the **all_xmltags** parameter:

```
CREATE INDEX bts_idx ON bts_100_tab(col2 bts_nvarchar_ops)
       USING bts(all_xmlattrs="yes") IN bts_sbspace1 ;
```

The index has these fields representing the attributes of the tags:

```
@callsign
@reg
```

Example 2: Combine all_xmlattrs and all_xmltags

The following CREATE INDEX statement uses both the **all_xmlattrs** and the **all_xmltags** parameters:

```
CREATE INDEX bts_idx ON bts_100_tab(col2 bts_nvarchar_ops)
       USING bts(all_xmlattrs="yes",
                 all_xmltags="yes") IN bts_sbspace1 ;
```

The index has these fields representing both the types of tags and the tag attributes:

```
@callsign
@reg
airplane
name
```

Example 3: Combine all_xmlattrs, all_xmltags, and xmlpath_processing

The following CREATE INDEX statement uses the **all_xmlattrs**, the **all_xmltags**, and the **xmlpath_processing** parameters:

```
CREATE INDEX bts_idx ON bts_100_tab(col2 bts_nvarchar_ops)
        USING bts(xmlpath_processing="yes",
                  all_xmlattrs="yes",
                  all_xmltags="yes") IN bts_sbspace1 ;
```

The index has these fields, representing the full paths of the tags and attributes:

```
/airplane
/airplane@callsign
/boat/name
/boat/name@reg
```

Example 4: Comparing all_xmltags to all_xmlattrs along with xmlpath_processing

The following CREATE INDEX statement uses the **all_xmltags** parameter with the **xmlpath_processing** parameter:

```
CREATE INDEX bts_idx ON bts_100_tab(col2 bts_nvarchar_ops)
        USING bts(xmlpath_processing="yes",
                  all_xmltags="yes") IN bts_sbspace1 ;
```

The index has these fields, representing the paths of the tags:

```
/airplane
/boat/name
```

The following CREATE INDEX statement uses the **all_xmlattrs** parameter with the **xmlpath_processing** parameter:

```
CREATE INDEX bts_idx ON bts_100_tab(col2 bts_nvarchar_ops)
        USING bts(xmlpath_processing="yes",
                 all_xmlattrs="yes") IN bts_sbspace1 ;
```

The index has these fields, representing the paths of the attributes:

```
/airplane@callsign
/boat/name@reg
```

The xmlpath_processing index parameter

Use the **xmlpath_processing** parameter to enable searches based on XML paths.

The **xmlpath_processing** parameter requires that you specify tags with the **xmltags** parameter or that you enable the **all_xmltags** or **all_xlmattrs** parameter.

When you enable **xmlpath_processing**, all the tags within the path are searched. Tags that are not within the path cannot be searched. If **xmlpath_processing** is not enabled only individual tags can be searched.

Full paths and relative paths in path processing

The XML path can be either a full path or a relative path.

Full paths

Full paths begins with a slash (/). If you use the **all_xmltags** parameter with **xmlpath_processing**, all of the full paths are indexed. You can index specific full or relative paths when you use the **xmltags** parameter.

Given the XML fragment:

```
<text>
<book>
 <title>Graph Theory</title>
 <author>Stewart</author>
 <date>January 14, 2006</date>
</book>
<text>
```

The following full XML paths can be processed with the **xmlpath_processing** parameter:

```
/text/book/title
/text/book/author
/text/book/date
```

Tip: If you have indexed a full path, include the initial slash (/) in the search predicate. For example:

```
bts_contains("/text/book/author:stewart")
```

Relative paths

Relative paths begin with text. You can specify one or more relative or full paths with the **xmltags** parameter.

Based on the preceding XML fragment, each of the following relative XML paths can be processed with the **xmlpath_processing** parameter:

```
text/book/title
text/book/author
text/book/date
book/title
book/author
book/date
title
author
date
```

The field is created from the first matching path for the values specified with the **xmltags** parameter.

You can create an index for the `book/title` and the `title` fields:

```
CREATE INDEX books_bts ON books(xml_data bts_lvarchar_ops)
using bts(xmltags="(book/title,title)",xmlpath_processing=”yes”)
IN bts_sbspace;
```

In that case, the index will contain only the first matching field, `book/title`. It will not contain a `title` field:

```
book/title:Graph Theory
```

To view the fields that you have indexed, use the bts_index_fields() function. See bts_index_fields() function on page 181.

Example: Index XML paths

Use XML path processing to restrict searches by paths.

Given the XML fragment:

```
<boat>
 <skipper>Captain Black</skipper>
 <boatname>The Queen Anne's Revenge</boatname>
 <alternate>
  <skipper>Captain Blue Beard</skipper>
 </alternate>
</boat>
```

Following are the possible XML paths and text values:

```
/boat/skipper:Captain Black
/boat/boathame:The Queen Anne's Revenge
/boat/alterate/skipper:Captain Blue Beard
```

To create an index for `boat/skipper` and `skipper`, use the statement:

```
CREATE INDEX boats_bts ON boats(xml_data bts_lvarchar_ops)
using bts(xmltags="(boat/skipper,skipper)",xmlpath_processing=”yes”)
IN bts_sbspace;
```

Each path is compared to the values specified by the **xmltags** parameter. The index then creates fields for the entire first matching path found for each **xmltags** value. In this example, the first path matches `boat/skipper`. The third path matches `skipper`. The index will contain two fields that can be searched:

```
/boat/skipper:Captain Black
/boat/alterate/skipper:Captain Blue Beard
```

The include_contents index parameter

Use the **include_contents** parameter to add the `contents` field to the index.

The **include_contents** parameter must be used with either the **xmltags** parameter specified or with the **all_xmltags** or **all_xmlattrs** parameter enabled.

When you do not use XML index parameters, XML documents are indexed as unstructured text in the `contents` field. When you specify the **xmltags** parameter or you enable the **all_xmltags** parameter, you can add the `contents` field to the index by enabling the **include_contents** parameter. This allows you to search the unstructured text in the contents field in addition to fields containing the tag or attribute text.

To view the fields that you have indexed, use the bts_index_fields() function. See bts_index_fields() function on page 181.

Example: Index XML tag values and XML tag names

Use the **include_contents** parameter to search both XML tag values and XML tag names.

Given the XML fragment:

```
<book>
 <title>Graph Theory</title>
 <author>Stewart</author>
 <date>January 14, 2006</date>
</book>
```

To create a **bts** index for all the tags as well as the XML tags in their unstructured form, use the statement:

```
CREATE INDEX book_bts ON books(xml_data bts_lvarchar_ops)
USING bts(all_xmltags="yes",include_contents="yes")
IN bts_sbspace;
```

The index will have four fields; one for each of the XML tags and one for the `contents` field:

```
title:graph theory
author:stewart
date:january 14, 2006
contents:<book> <title>Graph Theory</title> <author>Stewart</author>
 <date>January 14, 2006</date> </book>
```

The strip_xmltags index parameter

Use the **strip_xmltags** parameter to add the untagged values to the `contents` field in the index. Attribute values are also removed.

Unlike other XML index parameters, you can use the **strip_xmltags** parameter in a CREATE INDEX statement without specifying the **xmltags** parameter or enabling the **all_xmltags** parameter. In this case, the `contents` field is created automatically.

However, if you specify the **xmltags** parameter or if you enable the **all_xmltags** parameter, you must also enable the **include_contents** parameter.

To view the fields that you have indexed, use the bts_index_fields() function. See bts_index_fields() function on page 181.

Example: Index XML tag values in a separate field

Given the XML fragment:

```
<book>
 <title>Graph Theory</title>
 <author>Stewart</author>
 <date>January 14, 2006</date>
</book>
```

To create an index with the untagged values only, use the statement:

```
CREATE INDEX books_bts ON books(xml_data bts_lvarchar_ops)
USING bts(strip_xmltags="yes") IN bts_sbspace;
```

The index will contain a single `contents` field:

```
contents:Graph Theory Stewart January 14, 2006
```

To create an index that has XML tag fields as well as a field for the untagged values, use the statement:

```
CREATE INDEX book_bts ON books(xml_data bts_lvarchar_ops)
USING bts(all_xmltags="yes",include_contents="yes",strip_xmltags="yes")
IN bts_sbspace;
```

The index will contain XML tag fields as well as the untagged values in the `contents` field:

```
title:graph theory
author:stewart
date:january 14, 2006
contents:Graph Theory Stewart January 14, 2006
```

The include_namespaces index parameter

Use the **include_namespaces** parameter to index XML tags that include namespaces in the qualified namespace format *prefix*:*localpart*. For example:

```
<book:title></book:title>
```

The **include_namespaces** parameter must be used with either the **xmltags** parameter specified or with the **all_xmltags** parameter enabled.

When you enable the **include_namespaces** parameter and the data includes the namespace in the indexed tags, you must use the namespace prefix in your queries and escape each colon (:) with a backslash (\).

For example, to search for the text `Smith`, in the field `customer:name:`, use the format:

```
bts_contains("/customer\:name:Smith")
```

To view the fields that you have indexed, use the bts_index_fields() function. See bts_index_fields() function on page 181.

Example: Index namespaces in XML data

The following XML fragment contains the namespace `book:title`:

```
<book>
<book:title>Graph Theory</book:title>
<author>Stewart</author>
<date>January 14, 2006</date>
</book>
```

You can create a **bts** index with the **include_namespaces** parameter disabled as in the statement:

```
CREATE INDEX books_bts ON books(xml_data bts_lvarchar_ops)
USING bts(all_xmltags="yes",include_namespaces="no",xmlpath_processing="yes")
IN bts_sbspace;
```

In that case, the namespace prefix `book:` is ignored. The index will have the following fields.

```
/book/title:graph theory
/book/author:stewart
/book/date:january 14, 2006
```

Also, you can create a **bts** index with the **include_namespaces** parameter enabled, as in the statement:

```
CREATE INDEX books_bts ON books(xml_data bts_lvarchar_ops)
USING bts(all_xmltags="yes",include_namespaces="yes",xmlpath_processing="yes")
IN bts_sbspace;
```

In that case, the tag with the namespace `book:title` is the first field. The index has the following fields:

```
/book/book:title:graph theory
/book/author:stewart
/book/date:january 14, 2006
```

To search the field `/book/book:title:` for the text `theory`, use the search predicate:

```
bts_contains("/book/book\:title:theory")
```

When you specify tags with the **xmltags** parameter, you can index the tags with and without namespaces in different combinations using the **include_namespaces** parameter. For example, given the XML fragments:

```
<bsns:bookstore>
  <title> Marine Buyers' Guide </title>
  <bns2:title> Boat Catalog </bns2:title>
 </bsns:bookstore>

<bsns:bookstore>
  <bns1:title> Toy Catalog </bns1:title>
  <bns2:title> Wish Book </bns2:title>
 </bsns:bookstore>
```

To index only the `title` tag, use the format:

```
CREATE INDEX bookstore_bts ON bookstores(xml_data bts_lvarchar_ops)
USING bts(xmltag="(title)",include_namespaces="yes)
IN bts_sbspace;
```

Even though the **include_namespaces** parameter is enabled, the index will contain only one field because the fields `bns1:title` and `bns2:title` do not match the specified tag `title`.

If you want to index a namespace, include the namespace prefix in the specified tags. For example if you use the format:

```
CREATE INDEX bookstore_bts ON bookstores(xml_data bts_lvarchar_ops)
USING bts(xmltag="(title,bns1:title)",include_namespaces="yes)
IN bts_sbspace;
```

The index will contain the fields:

```
title: Marine Buyers' Guide
bns1:title: Toy Catalog
```

The include_subtag_text index parameter

Use the **include_subtag_text** parameter to index XML tags and subtags as one string. The **include_subtag_text** parameter is useful when you want to index text that has been formatted with bold `<b></b>` or italic `<i></i>` tags.

Use the **include_subtag_text** parameter with either the **xmltags** parameter specified or with the **all_xmltags** parameter enabled.

To view the fields that you have indexed, use the bts_index_fields() function. See bts_index_fields() function on page 181.

Example: Index subtags in XML data

You can use the **include_subtag_text** parameter to include the text within formatting tags in the indexed data.

Given the XML fragment:

```
<comment>
this
<bold> highlighted </bold>
text is very
<italic>
<bold>important</bold>
</italic>
to me
</comment>
```

If you create a **bts** index with the **include_subtag_text** parameter disabled:

```
CREATE INDEX comments_bts ON mylog(comment_data bts_lvarchar_ops)
USING bts(xmltags="(comment)",include_subtag_text="no") IN bts_sbspace;
```

The index will have three separate `comment` fields:

```
comment:this
comment:text is very
comment:to me
```

If you create a **bts** index with the **include_subtag_text** parameter enabled:

```
CREATE INDEX comments_bts ON mylog(comment_data bts_lvarchar_ops)
USING bts(xmltags="(comment)",include_subtag_text="yes") IN bts_sbspace;
```

All of the text is indexed in a single `comment` field:

```
comment:this highlighted text is very important to me
```

Basic text search analyzers

A text analyzer prescribes how text is indexed.

A text analyzer converts input text into tokens that are indexed.

Analyzers differ in the ways that they process the following text attributes:

- Letter case
- Stopwords
- Chinese, Japanese, and Korean characters
- Numbers and non-alphabetic characters
- White spaces

- Word stems
- Word pronunciation

If your needs are different than any of the basic text search analyzers, you can create a user-defined analyzer.

analyzer index parameter

When you create a **bts** index, you can include the analyzer index parameter to set the default analyzer and any specific analyzers for specific fields.

The analyzer index parameter

" **analyzer="** "

" { [*field* :] *analyzer* | ([*field* :] *analyzer*) | **file**: *directory/filename* | **table**:*table*. *column* } "

" " "

Table 24. Options for the analyzer index parameter

Element	Description
analyzer	The name of the analyzer. Possible values: • standard: Default. Processes alphabetic characters, special characters, and numbers with stopwords. • alnum: Processes strings of numbers and characters into tokens. • alnum+*characters*: Includes the specified characters in tokens. List characters without spaces. The maximum length of the character list is 128 bytes. • cjk: Processes Chinese, Japanese, and Korean text. Ignores surrogates. • cjk.ws: Processes Chinese, Japanese, and Korean text. Processes surrogates. • esoundex: Processes text into pronunciation codes. • keyword: Processes input text as a single token and adds trailing white spaces as necessary for fixed-length data type columns. • keyword.rt: Processes input text as a single token and removes trailing white spaces. • simple: Processes alphabetic characters only. Ignores stopword lists. • snowball: Processes text into stem words. • snowball.*language*: Processes text into stem words in the specified language. • soundex: Processes text into pronunciation codes. • stopword: Processes alphabetic characters only, except stopwords.

Table 24. Options for the analyzer index parameter (continued)

Element	Description
	• udr.*function_name*: Creates tokens according to the specified user-defined analyzer. • whitespace: Creates tokens that are based on white space only.
column	The name of the column that contains analyzer assignments.
directory	The path for the analyzer assignments file.
field	The XML tag, path, or the column name that is indexed.
filename	The name of the file that contains analyzer assignments.
table	The name of the table that contains analyzer assignments.

Usage

To use the same analyzer for all fields or columns that are indexed when you create the **bts** index, include the analyzer name without a field name. To use more than one analyzer, enclose multiple analyzer and field pairs in parentheses. To use one analyzer for most fields but other analyzers for specific fields, list the first analyzer without a field and the other analyzers with fields. The first analyzer is used for all fields except the ones that are explicitly listed with analyzer assignments.

You can specify the list of analyzers by field in a table column or in a file. The file or table must be readable by the user who creates the index. Separate the field name and analyzer pairs in the file or table by commas, white spaces, new lines, or a combination of those separators. The file or table becomes read-only when the index is created. If you want to add or change analyzer assignments, you must drop and re-create the index.

Examples

The following example creates a **bts** index on one column and uses the CJK analyzer:

```
CREATE INDEX desc_idx ON products (brands  bts_char_ops)
 USING bts (analyzer="cjk") IN sbsp1;
```

The following example creates a **bts** index on two XML fields and uses a different analyzer for each field:

```
CREATE INDEX boats_bts
ON boats(xml_data bts_lvarchar_ops)
USING bts
(
xmltags="(skipper,boatname)" ,
analyzer="(skipper:soundex,boatname:snowball)"
)
IN bts_sbspace;
```

Analyzer support for query and index options

The basic text search analyzer that you specify affects whether you can use stopwords or a thesaurus when you create an index and which query term modifiers you can use when you query text.

The following table shows which analyzers support query term modifiers, lowercase processing, stopwords, and a thesaurus.

Table 25. Analyzers and query term modifiers and index parameters

Analyzer	Word	Phrase	Wildcard	Fuzzy	Proximity	Range	Boolean	Lowercase	Stopwords	Thesaurus
Alnum	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
CJK	yes	yes	1	1	1	1	yes	yes	yes	yes
eSoundex	yes	yes	2	2	no	no	yes	yes	yes	no
Keyword	yes	yes	yes	yes	yes	no	yes	no	no	no
Simple	yes	yes	yes	yes	yes	yes	yes	yes	no	yes
Soundex	yes	yes	2	2	no	no	yes	yes	yes	no
Snowball	3	3	4	4	4	3	yes	yes	yes	yes
Standard	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
Stopword	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
User-defined analyzer	yes	5	5	5	5	5	yes	yes	yes	yes
Whitespace	yes	yes	yes	yes	yes	yes	yes	no	no	yes

1 = ISO Latin characters are supported.
2 = Must use the Soundex or eSoundex codes in the search terms.
3 = Depends on the stem word.
4 = The patterns must be on the stem word. The operation works on the stem word.
5 = Depends on the user-defined analyzer code.

Alnum analyzer

The Alnum analyzer is useful if you want to index words that contain numbers and other characters.

The Alnum analyzer processes text in the following ways:

- Indexes numbers as part of the word.
- Does not index stopwords.
- Converts alphabetic characters to lowercase.
- Treats as white space all non-alphanumeric characters unless the characters are included in the *characters* list. Non-alphanumeric characters include: #, %, $, @, &, :, ', (,) , -, _, \, and /.

Include a list of characters to index as part of words by using the alnum+*characters* syntax. List characters without spaces. The maximum length of the character list is 128 bytes.

Example

Examples

In these examples, the input string is shown on the first line and the resulting tokens are shown on the second line, each surrounded by square brackets.

In the following example, words that contain both numbers and letters are indexed together and special characters are treated as white spaces:

```
1002A 3234 abc123 xyz-abc lmn_opq
[1002a] [3234] [abc123] [xyz] [abc] [lmn] [opq]
```

In the following example, the analyzer index parameter is set to alnum+_-. The hyphen and underscore characters are indexed as part of words:

```
1002A 3234 abc123 xyz-abc lmn_opq
[1002a] [3234] [abc123] [xyz-abc] [lmn_opq]
```

CJK analyzer

The CJK analyzer processes Chinese, Japanese, and Korean characters into tokens that are indexed.

The CJK analyzer processes text characters in the following ways:

- Transforms the character sets to UTC-4. Half-width and full-width forms are converted so that they have equivalent characters. For example, fullwidth_digit_zero and digit_zero are treated as the same character.
- Indexes Chinese, Japanese, and Korean characters in overlapping pairs.
- Indexes Latin alphabetic, numeric, and the special characters _, +, and #.
- Stopwords are not indexed.
- Does not process supplementary code points if the analyzer name is cjk,
- Processes supplementary code points as surrogate pairs if the analyzer name is cjk.ws,

Example

Examples

In the following example, the first line shows the input string, in which C1, C2, C3 and C4 represent Chinese, Japanese, or Korean characters. The second line shows the resulting tokens, each surrounded by square brackets:

```
sailC1C2C3C4boat
[sail] [C1C2] [C2C3] [C3C4] [boat]
```

eSoundex analyzer

The eSoundex, or Extended Soundex, analyzer uses the Soundex algorithm to convert words into codes based on the English pronunciation of their consonants.

Vowel sounds are not included unless the vowel is the first letter of the word. The eSoundex analyzer is the similar to the Soundex analyzer except that it allows fewer or greater than four characters in its codes, depending on the length of the word. The eSoundex analyzer is useful if you want to search text based on how words sound. Because the text is converted to codes, you cannot perform proximity and range searches or specify a thesaurus.

The eSoundex analyzer processes text characters in the following ways:

- Stopwords are not indexed.
- Numbers and special characters are ignored.
- The colon (:) character is treated as a whitespace, so that characters on either side of it are considered separate words.

Example

Examples

In these examples, the input string is shown on the first line and the resulting tokens are shown on the second line, each surrounded by square brackets.

In the following example, the words "the" are not converted to tokens because they are stopwords and the rest of the words are converted to eSoundex codes that begin with the first letter of the word:

```
The Quick Brown Fox Jumped Over The Lazy Dog
[q2] [b65] [f2] [j513] [o16] [l2] [d2]
```

In the following example, the colon is treated as a whitespace and the backslash is ignored:

```
c:/informix
[c] [i51652]
```

In the following example, the ampersand is ignored:

```
XY&Z Corporation
[x2] [c61635]
```

In the following example, the e-mail address is considered one word:

```
xyz@example.com
[x2251425]
```

In the following example, numbers are ignored:

```
1abc 12abc abc1 abc12
[a12] [a12] [a12] [a12]
```

In the following examples, three words with the same stem word have different codes:

```
accept
[a213]
acceptable
[a21314]
acceptance
[a21352]
```

Keyword analyzer

The Keyword analyzer converts input text into a single token without alteration.

The Keyword analyzer is useful if you want to index single words exactly as they are, however, any type of input text is indexed. You cannot search a range or specify a thesaurus on text indexed by the Keyword analyzer.

The Keyword analyzer processes text characters in the following ways:

- Stopword lists are ignored. All words are indexed.
- Alphabetic characters are not converted to lowercase.
- Numeric and special characters are indexed.
- White spaces are indexed. Queries for text that includes white spaces must escape each white space by a backslash (\) character.
- If the analyzer name is keyword.rt, removes trailing white spaces during indexing and querying.
- If the analyzer name is keyword, indexes trailing white spaces.
    - For indexed columns that have fixed-length data types, the keyword analyzer adds white spaces as necessary to reach the column length. For example, if the text column is of type CHAR(6) and a string has three characters, `abc`, the string is indexed with three trailing white spaces, regardless of whether the string included one or more trailing white spaces: `abc ` . Queries require the correct number of escaped trailing white spaces: for example, `abc\ \ \ ` .
    - For indexed columns that have variable-length data types, any trailing white spaces that are included in the string are indexed. For example, if the text column is of type LVARCHAR, the string abc with one trailing white space is indexed as a different token from the string abc with two trailing white spaces. Queries require the correct number of escaped trailing white spaces: for example, `abc\ ` or `abc\ \ ` .

Example

Examples

In these examples, the input string is shown on the first line and the resulting tokens are shown on the second line, each surrounded by square brackets. The following examples show that the entire input string is preserved exactly as is:

```
The Quick Brown Fox Jumped Over The Lazy Dog
[The Quick Brown Fox Jumped Over The Lazy Dog]

-12 -.345 -898.2 -56. -
[-12 -.345 -898.2 -56. -]

XY&Z Corporation
[XY&Z Corporation]

xyz@example.com
[xyz@example.com]
```

The following query string searches for the string `The Quick Brown Fox Jumped Over The Lazy Dog`:

```
'The\ Quick\ Brown\ Fox\ Jumped\ Over\ The\ Lazy\ Dog'
```

Simple analyzer

The Simple analyzer converts text to tokens that contain only alphabetic characters.

The Simple analyzer is useful if you want to index every word and ignore non-alphabetical characters.

The Simple analyzer processes text characters in the following ways:

- Each word is processed into a separate token.
- Alphabetic characters are converted to lowercase.
- Numeric and special characters are treated as white spaces.
- Stopword lists are ignored. All words are indexed.

Because the Simple analyzer does not support stopwords, omit the word TO from range queries.

Example

Examples

In these examples, the input string is shown on the first line and the resulting tokens are shown on the second line, each surrounded by square brackets.

In the following example, every word is converted to a lowercase token:

```
The Quick Brown Fox Jumped Over The Lazy Dog
[the][quick][brown][fox] [jumped] [over] [the] [lazy] [dog]
```

In the following example, the @ symbol and period are treated as white spaces:

```
xyz@example.com
[xyz] [example] [com]
```

In the following example, numbers are not included in the tokens:

```
1abc 12abc abc1 abc12
[abc] [abc] [abc] [abc]
```

Soundex analyzer

The Soundex analyzer uses the Soundex algorithm to convert words into four-character codes based on the English pronunciation of their consonants.

Vowel sounds are not included unless the vowel is the first letter of the word. Additional sounds beyond the first four phonetic sounds are ignored. If a word has fewer than four phonetic sounds, zeros are used to complete the four-character codes. The Soundex analyzer is the similar to the eSoundex analyzer except that it uses four characters in its codes, regardless of the length of the word. The Soundex analyzer is useful if you want to search text based on how the beginnings of words sound. Because the text is converted to codes, you cannot perform proximity and range searches or specify a thesaurus.

The Soundex analyzer processes text characters in the following ways:

- Stopwords are not indexed.
- Numbers and special characters are ignored.
- The colon (:) character is treated as a whitespace, so that characters on either side of it are considered separate words.

Example

Examples

In these examples, the input string is shown on the first line and the resulting tokens are shown on the second line, each surrounded by square brackets. All codes consist of four characters.

In the following example, the words "the" are not converted to tokens because they are stopwords and the rest of the words are converted to Soundex codes that begin with the first letter of the word:

```
The Quick Brown Fox Jumped Over The Lazy Dog
[q200] [b650] [f200] [j513] [o160] [l200] [d200]
```

In the following example, the colon is treated as a whitespace and the backslash is ignored:

```
c:/informix
[c000] [i516]
```

In the following example, the ampersand is ignored:

```
XY&Z Corporation
[x200] [c616]
```

In the following example, the e-mail address is considered one word:

```
xyz@example.com
[x225]
```

In the following example, numbers are ignored:

```
1abc 12abc abc1 abc12
[a120] [a120] [a120] [a120]
```

In the following examples, three words with the same stem word have the same code:

```
accept
[a213]
acceptable
[a213]
acceptance
[a213]
```

Snowball analyzer

The Snowball analyzer converts words into language and code set specific stem words.

The Snowball analyzer is similar to the Standard analyzer except that is converts words to stem words.

The Snowball analyzer processes text characters in the following ways:

- Converts words to stem word tokens.
- Stopwords are not indexed.
- Converts alphabetical characters to lower case.
- Ignores colons, #, %, $, parentheses, and slashes.
- Indexes underscores, hyphens, @, and & symbols when they are part of words or numbers.
- Separately indexes numbers and words if numbers appear at the beginning of a word.
- Indexes numbers as part of the word if they are within or at the end of the word.
- Indexes apostrophes if they are in the middle of a word, but removes them if they are at the beginning or end of a word.
- Ignores an apostrophe followed by the letter s at the end of a word.

By default, the Snowball analyzer uses the language and code set that is specified by the **DB_LOCALE** environment variable. You can specify a different language for the Snowball analyzer by appending the language name or synonym to the Snowball analyzer name in the CREATE INDEX statement: snowball.*language*.

- Danish, da, dan
- Dutch, nl nld, dut
- English, en, eng
- Porter, por (the original English stemmer)
- Finnish, fi, fin
- French, fr, fra, fre
- German, de, deu, ger
- Italian, it, ita
- Norwegian, no, nor
- Portuguese, pt
- Spanish, es, esl, spa
- Swedish, sv, swe

The Snowball analyzer supports the 8859-1 code set.

Example

Examples

In these examples, the input string is shown on the first line and the resulting tokens are shown on the second line, each surrounded by square brackets. These examples use the English language, specified by the analyzer="snowball.en" index parameter. For examples of how the Snowball analyzer uses word stemming in languages other than English, see the Snowball web site at http://snowball.tartarus.org.

In the following example, stopwords are removed, the words are converted to lower case, and the word "lazy" is converted to its stem word:

```
The Quick Brown Fox Jumped Over The Lazy Dog
[quick] [brown] [fox] [jump] [over] [lazi] [dog]
```

In the following example, the apostrophe at the beginning of a word and the apostrophe followed by an s are ignored, but the apostrophe in the middle of a word is indexed:

```
Prequ'ile Mark's 'cause
[prequ'ile] [mark] [cause]
```

In the following example, the colon and backslash are ignored:

```
c:/informix
[c] [informix]
```

In the following example, the ampersand is indexed as part of the company name:

```
XY&Z Corporation
[xy&z] [corpor]
```

In the following example, the e-mail address is indexed as is:

```
xyz@example.com
[xyz@example.com]
```

In the following example, the three different words are indexed with the same stem word:

```
accept
[accept]

acceptable
[accept]

acceptance
[accept]
```

Standard analyzer

The Standard analyzer removes stopwords and indexes words, numbers, and some special characters. The Standard analyzer is the default analyzer.

The Standard analyzer processes text characters in the following ways:

- Stopwords are not indexed.
- Converts alphabetical characters to lower case.
- Ignores colons, #, %, $, parentheses, hyphens, and slashes.
- Indexes underscores, @, and & symbols when they are part of words or numbers.
- Separately indexes number and words if numbers appear at the beginning of a word.
- Indexes numbers as part of the word if they are within or at the end of the word.
- Indexes apostrophes if they are in the middle of a word, but removes them if they are at the beginning or end of a word.
- Ignores an apostrophe followed by the letter s at the end of a word.

Example

Examples

In these examples, the input string is shown on the first line and the resulting tokens are shown on the second line, each surrounded by square brackets.

In the following example, stopwords are removed and the words are converted to lower case:

```
The Quick Brown Fox Jumped Over The Lazy Dog
[quick] [brown] [fox] [jumped] [over] [lazy] [dog]
```

In the following example, the apostrophe at the beginning of a word and the apostrophe followed by an s are ignored, but the apostrophe in the middle of a word is indexed:

```
Prequ'ile Mark's 'cause
[prequ'ile] [mark] [cause]
```

In the following example, the colon and backslash are ignored:

```
c:/informix
[c] [informix]
```

In the following example, the ampersand is indexed as part of the company name:

```
XY&Z Corporation
[xy&z] [corporation]
```

In the following example, the e-mail address is indexed as is:

```
xyz@example.com
[xyz@example.com]
```

In the following example, numbers at the beginning of the words are separated into different tokens, while numbers at the end of words are included in a single token:

```
1abc 12abc abc1 abc12
[1] [abc] [12] [abc] [abc1] [abc12]
```

Stopword analyzer

The Stopword analyzer removes stopwords and converts text to tokens that contain only alphabetic characters.

The Stopword analyzer is useful if you want to remove stopwords and ignore non-alphabetical characters.

The Stopword analyzer processes text characters in the following ways:

- Each word is processed into a separate token.
- Alphabetic characters are converted to lowercase.
- Numeric and special characters are treated as white spaces.
- Stopwords are not indexed.

Example

Examples

In these examples, the input string is shown on the first line and the resulting tokens are shown on the second line, each surrounded by square brackets.

In the following example, stopwords are removed and the letters are converted to lowercase:

```
The Quick Brown Fox Jumped Over The Lazy Dog
[quick] [brown] [fox] [jumped] [over] [lazy] [dog]
```

In the following example, the @ symbol and period are treated as white spaces:

```
xyz@example.com
[xyz] [example] [com]
```

In the following example, numbers are not included in the tokens:

```
1abc 12abc abc1 abc12
[abc] [abc] [abc] [abc]
```

User-defined analyzer

A user-defined analyzer processes text into tokens according to a user-defined function.

You can write a user-defined function to process text into tokens according to your needs. Use udr.*function_name* as the analyzer name with the analyzer option when you create a basic text search index.

Example

Examples

The following function, which is written in C, processes alphabetical and numeric characters into tokens and ignores all special characters except underscore (_):

```
/*ARGSUSED*/
UDREXPORT
mi_lvarchar* tokenize_alnum(
    mi_lvarchar*    string,
    MI_FPARAM*      fparam)
{
    mi_integer      status = MI_OK;
    mi_lvarchar*    rtn = NULL;
    gl_mchar_t*     src = NULL;
    gl_mchar_t*     tgt = NULL;
    mi_integer      token = 0;
    gl_mchar_t*     s;
    gl_mchar_t*     r;

    ifx_gl_init();
    if (((src = (gl_mchar_t*)mi_lvarchar_to_string(string)) == NULL) ||
        ((tgt = (gl_mchar_t*)mi_alloc((strlen(src)*4)+1)) == NULL)) {
        status = MI_ERROR;
        goto cleanup;
    }
    s = src;
    r = tgt;
```

```
    while ((s != NULL) && (*s != '\0')) {
        if ((ifx_gl_ismalnum(s, IFX_GL_NO_LIMIT)) || (*s == '_')) {
            if (!token) {
                if (r != tgt) *r++ = ' ';
                *r++ = '[';
                token = 1;
            }
            ifx_gl_mbsncpy(r, s, IFX_GL_NULL, 1);
            r = ifx_gl_mbsnext(r, IFX_GL_NO_LIMIT);
        }
        else {
            if (token) {
                *r++ = ']';
                token = 0;
            }
        }
        s = ifx_gl_mbsnext(s, IFX_GL_NO_LIMIT);
    }
    if (token) *r++ = ']';
    *r = '\0';
    if ((rtn = mi_string_to_lvarchar((char*)tgt)) == NULL) {
        status = MI_ERROR;
        goto cleanup;
    }
cleanup:
    if ((status != MI_OK) &&
        (rtn != NULL)) {
        mi_var_free(rtn);
        rtn = NULL;
    }
    if (tgt != NULL) mi_free(tgt);
    if (src != NULL) mi_free(src);
    if (rtn == NULL) mi_fp_setreturnisnull(fparam, 0, MI_TRUE);
    return rtn;
}
```

The following statement registers the function so that the database server can use it:

```
CREATE FUNCTION tokenize_alnum (lvarchar)
    RETURNS lvarchar
    WITH (NOT VARIANT)
    EXTERNAL NAME "$ONEDB_HOME/extend/myblade/myblade.bld(tokenize_alnum)"
        LANGUAGE C;
```

When an index is created with the analyzer="udr.tokenize_alnum" option, the following example shows that no special characters except the underscore are indexed:

```
quick! #$%&^^$## Brown fox under_score
[quick] [Brown] [fox] [under_score]
```

Whitespace analyzer

The Whitespace analyzer processes characters into tokens based on whitespaces. All characters between whitespaces are indexed without alteration.

The Whitespace analyzer processes text characters in the following ways:

- Stopword lists are ignored. All words are indexed.
- Does not change letter case.
- Indexes numbers and special characters.

Because the Whitespace analyzer does not support stopwords, omit the word TO from range queries.

Example

Examples

In the following examples, the input text is shown on the first line and the resulting indexed tokens, which are surrounded by square brackets, are shown on the second line.

In the following example, all words are indexed exactly as they are:

```
The Quick Brown Fox Jumped Over The Lazy Dog
[The] [Quick] [Brown][Fox] [Jumped][Over] [The] [Lazy] [Dog]
```

In the following example, all numbers and special characters are indexed:

```
-12 -.345 -898.2 -56. -
[-12] [-.345] [-898.2] [-56.] [-]
```

In the following example, the e-mail address is indexed as one token:

```
xyz@example.com
[xyz@example.com]
```

Basic text search functions

You can use basic text search functions to provide information about **bts** indexes, compact bts indexes, and configure tracing.

bts_index_compact() function

The bts_index_compact() function deletes all documents from the **bts** index that are marked as deleted.

Syntax

```
bts_index_compact('index_name')
```

index_name

The name of the **bts** index for which you want to delete rows.

Usage

Use the bts_index_compact() function to delete documents from a **bts** index that was created with the default deletion mode parameter of **delete="deferred"**. The bts_index_compact() function releases space in the index by immediately deleting the rows marked as deleted. The index is unavailable while it is rewritten. Optionally, you can include the index storage space path and file name, the database name, and the owner name in addition to the index name, separated by forward slash (/) characters.

Documents marked as deleted can also be deleted with the oncheck utility. For oncheck syntax and information about optimizing the **bts** index, see delete index parameter on page 124.

Return codes

t

The operation was successful.

f

The operation was unsuccessful.

Example

The following example compacts the **bts** index `desc_idx`:

```
EXECUTE FUNCTION bts_index_compact('desc_idx');
```

bts_index_fields() function

The bts_index_fields() function returns the list of indexed field names in the **bts** index.

Syntax

```
bts_index_fields('index_name')
```

index_name

The name of the **bts** index.

Usage

Use the bts_index_fields() function to identify searchable fields in the **bts** index. Optionally, you can include the index storage space path and file name, the database name, and the owner name in addition to the index name, which is separated by forward slash (/) characters.

The bts_index_fields() function returns one default field that is called `contents` unless any of the following conditions are true:

- The index is a composite index that has each column that is indexed separately because the index definition includes the query_default_field="*" index parameter. The bts_index_fields() function returns the names of the indexed columns.
- The index contains XML tags because the index definition includes the **all_xmltags** or **xmltags** index parameter. The bts_index_fields() function returns the indexed tags. If the include_contents index parameter is included in the index definition, the bts_index_fields() function also returns the `contents` field.
- The index contains JSON field name-value pairs because the index definition includes the all_json_names or json_names index parameter. The bts_index_fields() function returns the indexed field names. If the include_contents index parameter is included in the index definition, the bts_index_fields() function also returns the `contents` field.

When you specify tags with the **xmltags** parameter, the bts_index_fields() function returns only field names for tags that exist in the indexed column. However, if later you add a row that contains the specified tag name, the field name for that tag appears in the output.

The bts_index_fields() function returns the field names in alphabetical order.

Example

Example: Unstructured index

The following statement creates an unstructured index:

```
CREATE INDEX desc_idx ON products (brands  bts_char_ops)
 USING bts IN sbsp1;
```

The bts_index_fields() function returns the default field: `contents`.

Examples: Structured indexes on an XML document

These examples are based on the following XML fragment:

```
<boat>
 <skipper>Captain Jack</skipper>
 <boatname>Black Pearl</boatname>
</boat>
```

The following statement indexes the specified XML tags:

```
CREATE INDEX boats_bts ON boats(xml_data bts_lvarchar_ops)
USING bts(xmltags="(skipper,boatname,crew)") IN bts_sbspace;
```

The bts_index_fields() function returns the following field names:

```
boatname
skipper
```

The field name for the tag `crew` is not returned because it does not exist in the XML fragment example.

The following statement indexes all tags and paths:

```
CREATE INDEX boats_bts ON boats(xml_data bts_lvarchar_ops)
USING bts(all_xmltags="yes",xmlpath_processing="yes")
IN bts_sbspace;
```

The bts_index_fields() function returns field names that include full paths:

```
/boat/boatname
/boat/skipper
```

The following statement indexes all tags and includes the `contents` field:

```
CREATE INDEX boats_bts ON boats(xml_data bts_lvarchar_ops)
USING bts(all_xmltags="yes",include_contents="yes")
IN bts_sbspace;
```

The bts_index_fields() function returns the following fields:

```
boatname
contents
skipper
```

Example

Examples: Structured indexes on a JSON document

These examples are based on the following JSON document:

```
{ "person" : {
     "givenname" : "Jim",
     "surname" : "Flynn",
     "age" : 29,
     "cars" : [ "dodge", "olds" ],
     "parents":[
        { "givenname" : "Slim",
          "surname" : "Flynn",
          "surname" : "Kim" }
     ]
}
```

The following statement indexes all field name-value pairs:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(all_json_names="yes");
```

The bts_index_fields() function returns the following fields:

```
age
cars
givenname
surname
```

The following statement indexes all field name-value pairs and includes the `contents` field:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(all_json_names="yes",
                 include_contents="yes");
```

The bts_index_fields() function returns the following fields:

```
age
cars
contents
givenname
surname
```

The following statement indexes a single field:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(json_names="surname");
```

The bts_index_fields() function returns the following field:

```
surname
```

The following statement indexes the specified field names and paths:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(
                 json_names="(parents.surname,
                              person.givenname)",
                 json_path_processing="yes"
        );
```

The bts_index_fields() function returns the following fields:

```
person.given name
parents.surname
```

bts_release() function

The **bts_release()** function provides the internal release version number of the basic text search engine.

Syntax

```
bts_release()
```

Usage

Use the bts_release() function if HCL Software Support asks you for the basic text search version number.

Return codes

This function returns the name and release version number of the basic text search engine.

Example

Example output:

```
BTS 3.00 Compiled on Wed Jan 19 11:25:52 CDT 2011
```

bts_tracefile() function

The bts_tracefile() function specifies the location where the trace file is written. Use this function together with the bts_tracelevel() function to trace basic text search-related events.

Syntax

```
bts_tracefile(filename )
```

filename

The full path and name of the file to which trace information is appended. The file must be writable by user **informix**. If no file name is provided, a standard *session_id*.trc file is placed in the $ONEDB_HOME/tmp directory.

Usage

Use the bts_tracefile() function to troubleshoot events related to the basic text searches.

For the syntax for bts_tracelevel(), see bts_tracelevel() function on page 185.

For more details about tracing, see the *HCL OneDB™ Guide to SQL: Reference*.

Example

The following example specifies a trace log named `bts_select.log` in the `/tmp` directory:

```
EXECUTE FUNCTION bts_tracefile('/tmp/bts_select.log');
```

bts_tracelevel() function

The bts_tracelevel() function sets the level of tracing. Use this function together with the bts_tracefile() function to trace basic text search-related events.

Syntax

```
bts_tracelevel(level)
```

level

The level of tracing output:

1

UDR entry points.

10

UDR entry points and lower-level calls.

20

Trace information and small events.

100

Memory resource tracing (very verbose).

If you enter a value from 1-9, it is treated as level 1, a value between 10 and 19 is treated as level 10, a value between 20 and 99 is treated as level 20. A value greater than or equal to 100 is treated as level 100.

Usage

Use the bts_tracelevel() function to troubleshoot events related to the basic text search extension.

For the syntax for bts_tracefile(), see bts_tracefile() function on page 184.

For more details about tracing, see the *HCL OneDB™ Guide to SQL: Reference*.

Example

The following example specifies a trace file, sets the trace level to 20, and then performs a SELECT statement, which generates a tracing log:

```
EXECUTE FUNCTION bts_tracefile('/tmp/bts_select.log');
EXECUTE FUNCTION bts_tracelevel(20);
SELECT * FROM vessels WHERE bts_contains(xml_info, 'boatname:black');
```

The following might be the contents of the tracing log for trace level 20. The number `32` is the trace session number.

```
============================================================

Tracing session: 32 on 03/26/2009

09:21:11  BTS[32] bts_tracelevel_set: exit  (level = 20, status = 0)
09:21:11  BTS[32] bts_am_cost: entry
09:21:11  BTS[32] bts_am_cost: exit  (status = 0, cost = 0.500000)
09:21:11  BTS[32] bts_am_open: entry
09:21:11  BTS[32] bts_init: entry
09:21:11  BTS[32] bts_lock_try: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: exit  (lock name: 'BTS_LOCK-EVP', status = 0)
09:21:11  BTS[32] bts_lock_try: exit  (status = 0)
09:21:11  BTS[32] bts_cl_init: entry (bts_cl_init_value = 0)
09:21:11  BTS[32] bts_cl_init_restore: entry
09:21:11  BTS[32] bts_cl_init_setup: entry
09:21:11  BTS[32] bts_cl_init_setup: exit  (status = 0)
09:21:11  BTS[32] bts_cl_init_restore: exit  (status = 0)
09:21:11  BTS[32] bts_cl_init: exit  (bts_cl_init_value = 1, status = 0)
09:21:11  BTS[32] bts_gls_init: entry
09:21:11  BTS[32] bts_gls_init: exit  (status = 0)
09:21:11  BTS[32] bts_evp_check: entry
09:21:11  BTS[32] bts_evp_check: exit  (status = 0)
09:21:11  BTS[32] bts_auto_trace: (skipped)
09:21:11  BTS[32] bts_init: exit  (status = 0)
09:21:11  BTS[32] bts_am_spacename: entry
09:21:11  BTS[32] bts_am_spacename: exit  (spacename = 'bts_sbspace1', status = 0)
09:21:11  BTS[32] bts_am_space: entry
09:21:11  BTS[32] bts_am_sbspace: entry
09:21:11  BTS[32] bts_am_sbspace: exit  (rtn = '/ashworth/vessels_bts/1048885', status = 0)
09:21:11  BTS[32] bts_am_space: exit  (rtn = '/ashworth/vessels_bts/1048885', status = 0)
09:21:11  BTS[32] bts_hdr_check: entry
09:21:11  BTS[32] bts_hdr_check: (hdr_status mask = 00000000)
09:21:11  BTS[32] bts_hdr_check: exit  (status = 0)
09:21:11  BTS[32] bts_lock_try: entry (name = '/ashworth/vessels_bts/1048885')
09:21:11  BTS[32] bts_lock_try: exit  (status = 0)
09:21:11  BTS[32] bts_am_params_read: entry
09:21:11  BTS[32] bts_am_params_canonical_maps_setup: entry
09:21:11  BTS[32] bts_am_params_canonical_maps_setup: (expand = 1)
09:21:11  BTS[32] bts_am_params_canonical_maps_setup: exit  (status = 0)
09:21:11  BTS[32] bts_am_params_read: exit  (status = 0)
09:21:11  BTS[32] bts_lock_release: entry (name = '/ashworth/vessels_bts/1048885')
09:21:11  BTS[32] bts_lock_release: exit  (status = 0)
09:21:11  BTS[32] bts_am_open: (open set_size 256)
09:21:11  BTS[32] bts_xact_register: entry
09:21:11  BTS[32] bts_xact_register: (XACT: named_memory(BTS_XACT_20))
09:21:11  BTS[32] bts_xact_register: (new savepoint: 1-1 (first))
```

```
09:21:11  BTS[32] bts_xact_register: (register savepoint callback)
09:21:11  BTS[32] bts_xact_register: (register end_stmt callback)
09:21:11  BTS[32] bts_xact_register: (register end_xact callback)
09:21:11  BTS[32] bts_xact_register: (register post_xact callback)
09:21:11  BTS[32] bts_xact_register: exit  (status = 0)
09:21:11  BTS[32] bts_xact_log_params: entry
09:21:11  BTS[32] bts_xact_init_bxt: exit  (status = 0)
09:21:11  BTS[32] bts_am_params_copy: exit  (status = 0)
09:21:11  BTS[32] bts_xact_log_params: (XACT: sbspace(bts_sbspace1))
09:21:11  BTS[32] bts_xact_log_params: (XACT: space_type(1))
09:21:11  BTS[32] bts_xact_log_params: exit  (status = 0)
09:21:11  BTS[32] bts_fini: entry (errcode = 0)
09:21:11  BTS[32] bts_cl_fini: entry (bts_cl_init_value = 1)
09:21:11  BTS[32] bts_cl_init_clear: entry
09:21:11  BTS[32] bts_cl_init_clear: exit  (status = 0)
09:21:11  BTS[32] bts_cl_fini: exit  (bts_cl_init_value = 0, status = 0)
09:21:11  BTS[32] bts_lock_release: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: exit  (lock name: 'BTS_LOCK-EVP', status = 0)
09:21:11  BTS[32] bts_lock_release: exit  (status = 0)
09:21:11  BTS[32] bts_fini: exit  (status = 0)
09:21:11  BTS[32] bts_am_open: exit  (status = 0)
09:21:11  BTS[32] bts_am_beginscan: entry
09:21:11  BTS[32] bts_am_userdata_get: entry
09:21:11  BTS[32] bts_am_spacename: entry
09:21:11  BTS[32] bts_am_spacename: exit  (spacename = 'bts_sbspace1', status = 0)
09:21:11  BTS[32] bts_am_userdata_get: (target = '/ashworth/vessels_bts/1048885')
09:21:11  BTS[32] bts_am_userdata_get: exit  (status = 0)
09:21:11  BTS[32] bts_am_beginscan: (target = '/ashworth/vessels_bts/1048885')
09:21:11  BTS[32] bts_am_literal: entry
09:21:11  BTS[32] bts_am_literal_size: entry
09:21:11  BTS[32] bts_am_literal_size: exit  (status = 0)
09:21:11  BTS[32] bts_am_literal_cat: entry
09:21:11  BTS[32] bts_am_literal_cat: exit  (status = 0)
09:21:11  BTS[32] bts_am_literal: (literal is 'boatname:black')
09:21:11  BTS[32] bts_am_literal: exit  (status = 0)
09:21:11  BTS[32] bts_am_beginscan: (literal = 'boatname:black')
09:21:11  BTS[32] bts_am_beginscan: (rows = 256, score needed = 'no')
09:21:11  BTS[32] bts_am_beginscan: exit  (status = 0)
09:21:11  BTS[32] bts_am_getnext: entry
09:21:11  BTS[32] bts_init: entry
09:21:11  BTS[32] bts_lock_try: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: exit  (lock name: 'BTS_LOCK-EVP', status = 0)
09:21:11  BTS[32] bts_lock_try: exit  (status = 0)
09:21:11  BTS[32] bts_cl_init: entry (bts_cl_init_value = 0)
09:21:11  BTS[32] bts_cl_init_restore: entry
09:21:11  BTS[32] bts_cl_init_restore: exit  (status = 0)
09:21:11  BTS[32] bts_cl_init: exit  (bts_cl_init_value = 1, status = 0)
09:21:11  BTS[32] bts_gls_init: entry
09:21:11  BTS[32] bts_gls_init: exit  (status = 0)
09:21:11  BTS[32] bts_evp_check: entry
```

```
09:21:11  BTS[32] bts_evp_check: exit  (status = 0)
09:21:11  BTS[32] bts_auto_trace: (skipped)
09:21:11  BTS[32] bts_init: exit  (status = 0)
09:21:11  BTS[32] bts_lock_try: entry (name = '/ashworth/vessels_bts/1048885')
09:21:11  BTS[32] bts_lock_try: exit  (status = 0)
09:21:11  BTS[32] bts_cl_query: entry
09:21:11  BTS[32] bts_cl_query_setup: entry
09:21:11  BTS[32] bts_xact_get_cl_cb: entry
09:21:11  BTS[32] bts_xact_get_cl_cb: exit  (status = 0)
09:21:11  BTS[32] bts_cl_query_parse: entry
09:21:11  BTS[32] bts_cl_query_dump: entry
09:21:11  BTS[32] bts_cl_query_dump: (max clause count = 1024)
09:21:11  BTS[32] bts_cl_query_dump: (query default operator = '0' (or))
09:21:11  BTS[32] bts_cl_query_dump: (query = 'boatname:black')
09:21:11  BTS[32] bts_cl_query_dump: (keyfield = 'boatname')
09:21:11  BTS[32] bts_cl_query_dump: exit  (status = 0)
09:21:11  BTS[32] bts_cl_query_parse: exit  (status = 0)
09:21:11  BTS[32] bts_cl_query_setup: exit  (status = 0)
09:21:11  BTS[32] bts_cl_query_parse: entry
09:21:11  BTS[32] bts_cl_query_parse: exit  (status = 0)
09:21:11  BTS[32] bts_cl_query: exit  (status = 0)
09:21:11  BTS[32] bts_am_getnext: (return 0 (0) fragid = 1048884, rowid = 257)
09:21:11  BTS[32] bts_lock_release: entry (name = '/ashworth/vessels_bts/1048885')
09:21:11  BTS[32] bts_lock_release: exit  (status = 0)
09:21:11  BTS[32] bts_fini: entry (errcode = 0)
09:21:11  BTS[32] bts_cl_fini: entry (bts_cl_init_value = 1)
09:21:11  BTS[32] bts_cl_init_clear: entry
09:21:11  BTS[32] bts_cl_init_clear: exit  (status = 0)
09:21:11  BTS[32] bts_cl_fini: exit  (bts_cl_init_value = 0, status = 0)
09:21:11  BTS[32] bts_lock_release: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: exit  (lock name: 'BTS_LOCK-EVP', status = 0)
09:21:11  BTS[32] bts_lock_release: exit  (status = 0)
09:21:11  BTS[32] bts_fini: exit  (status = 0)
09:21:11  BTS[32] bts_am_getnext: exit  (status = 1)
09:21:11  BTS[32] bts_am_getnext: entry
09:21:11  BTS[32] bts_init: entry
09:21:11  BTS[32] bts_lock_try: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: exit  (lock name: 'BTS_LOCK-EVP', status = 0)
09:21:11  BTS[32] bts_lock_try: exit  (status = 0)
09:21:11  BTS[32] bts_cl_init: entry (bts_cl_init_value = 0)
09:21:11  BTS[32] bts_cl_init_restore: entry
09:21:11  BTS[32] bts_cl_init_restore: exit  (status = 0)
09:21:11  BTS[32] bts_cl_init: exit  (bts_cl_init_value = 1, status = 0)
09:21:11  BTS[32] bts_gls_init: entry
09:21:11  BTS[32] bts_gls_init: exit  (status = 0)
09:21:11  BTS[32] bts_evp_check: entry
09:21:11  BTS[32] bts_evp_check: exit  (status = 0)
09:21:11  BTS[32] bts_auto_trace: (skipped)
09:21:11  BTS[32] bts_init: exit  (status = 0)
09:21:11  BTS[32] bts_lock_try: entry (name = '/ashworth/vessels_bts/1048885')
```

```
09:21:11  BTS[32] bts_lock_try: exit  (status = 0)
09:21:11  BTS[32] bts_cl_query: entry
09:21:11  BTS[32] bts_cl_query_next: entry
09:21:11  BTS[32] bts_cl_query_parse: entry
09:21:11  BTS[32] bts_cl_query_parse: exit  (status = 0)
09:21:11  BTS[32] bts_cl_query_next: exit  (status = 0)
09:21:11  BTS[32] bts_cl_query_parse: entry
09:21:11  BTS[32] bts_cl_query_parse: exit  (status = 0)
09:21:11  BTS[32] bts_cl_query: exit  (status = 0)
09:21:11  BTS[32] bts_lock_release: entry (name = '/ashworth/vessels_bts/1048885')
09:21:11  BTS[32] bts_lock_release: exit  (status = 0)
09:21:11  BTS[32] bts_fini: entry (errcode = 0)
09:21:11  BTS[32] bts_cl_fini: entry (bts_cl_init_value = 1)
09:21:11  BTS[32] bts_cl_init_clear: entry
09:21:11  BTS[32] bts_cl_init_clear: exit  (status = 0)
09:21:11  BTS[32] bts_cl_fini: exit  (bts_cl_init_value = 0, status = 0)
09:21:11  BTS[32] bts_lock_release: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: exit  (lock name: 'BTS_LOCK-EVP', status = 0)
09:21:11  BTS[32] bts_lock_release: exit  (status = 0)
09:21:11  BTS[32] bts_fini: exit  (status = 0)
09:21:11  BTS[32] bts_am_getnext: exit  (status = 0)
09:21:11  BTS[32] bts_xact_end_stmt: entry
09:21:11  BTS[32] bts_xact_bxh_init: entry
09:21:11  BTS[32] bts_xact_bxh_init: (XACT: named_memory(BTS_XACT_20))
09:21:11  BTS[32] bts_xact_bxh_init: exit  (status = 0, bxh = 0x53661ce8)
09:21:11  BTS[32] bts_init: entry
09:21:11  BTS[32] bts_lock_try: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: exit  (lock name: 'BTS_LOCK-EVP', status = 0)
09:21:11  BTS[32] bts_lock_try: exit  (status = 0)
09:21:11  BTS[32] bts_cl_init: entry (bts_cl_init_value = 0)
09:21:11  BTS[32] bts_cl_init_restore: entry
09:21:11  BTS[32] bts_cl_init_restore: exit  (status = 0)
09:21:11  BTS[32] bts_cl_init: exit  (bts_cl_init_value = 1, status = 0)
09:21:11  BTS[32] bts_gls_init: entry
09:21:11  BTS[32] bts_gls_init: exit  (status = 0)
09:21:11  BTS[32] bts_evp_check: entry
09:21:11  BTS[32] bts_evp_check: exit  (status = 0)
09:21:11  BTS[32] bts_auto_trace: (skipped)
09:21:11  BTS[32] bts_init: exit  (status = 0)
09:21:11  BTS[32] bts_xact_end_stmt: (procesing current_stmt: 1)
09:21:11  BTS[32] bts_xact_process: entry
09:21:11  BTS[32] bts_xact_process: (process: NORMAL_END)
09:21:11  BTS[32] bts_xact_process: (process end_stmt: 1)
09:21:11  BTS[32] bts_xact_process: (current savepoint is 1-1)
09:21:11  BTS[32] bts_lock_try: entry (name = '/ashworth/vessels_bts/1048885')
09:21:11  BTS[32] bts_lock_try: exit  (status = 0)
09:21:11  BTS[32] bts_lock_release: entry (name = '/ashworth/vessels_bts/1048885')
09:21:11  BTS[32] bts_lock_release: exit  (status = 0)
09:21:11  BTS[32] bts_xact_process: exit  (status = 0)
09:21:11  BTS[32] bts_xact_end_stmt: (new stmt: 2)
```

```
09:21:11  BTS[32] bts_fini: entry (errcode = 0)
09:21:11  BTS[32] bts_cl_fini: entry (bts_cl_init_value = 1)
09:21:11  BTS[32] bts_cl_init_clear: entry
09:21:11  BTS[32] bts_cl_init_clear: exit  (status = 0)
09:21:11  BTS[32] bts_cl_fini: exit  (bts_cl_init_value = 0, status = 0)
09:21:11  BTS[32] bts_lock_release: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: exit  (lock name: 'BTS_LOCK-EVP', status = 0)
09:21:11  BTS[32] bts_lock_release: exit  (status = 0)
09:21:11  BTS[32] bts_fini: exit  (status = 0)
09:21:11  BTS[32] bts_xact_end_stmt: exit  (status = 0, state = 0)
09:21:11  BTS[32] bts_am_endscan: entry
09:21:11  BTS[32] bts_init: entry
09:21:11  BTS[32] bts_lock_try: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: exit  (lock name: 'BTS_LOCK-EVP', status = 0)
09:21:11  BTS[32] bts_lock_try: exit  (status = 0)
09:21:11  BTS[32] bts_cl_init: entry (bts_cl_init_value = 0)
09:21:11  BTS[32] bts_cl_init_restore: entry
09:21:11  BTS[32] bts_cl_init_restore: exit  (status = 0)
09:21:11  BTS[32] bts_cl_init: exit  (bts_cl_init_value = 1, status = 0)
09:21:11  BTS[32] bts_gls_init: entry
09:21:11  BTS[32] bts_gls_init: exit  (status = 0)
09:21:11  BTS[32] bts_evp_check: entry
09:21:11  BTS[32] bts_evp_check: exit  (status = 0)
09:21:11  BTS[32] bts_auto_trace: (skipped)
09:21:11  BTS[32] bts_init: exit  (status = 0)
09:21:11  BTS[32] bts_lock_try: entry (name = '/ashworth/vessels_bts/1048885')
09:21:11  BTS[32] bts_lock_try: exit  (status = 0)
09:21:11  BTS[32] bts_cl_query_end: entry
09:21:11  BTS[32] bts_cl_query_parse: entry
09:21:11  BTS[32] bts_cl_query_parse: exit  (status = 0)
09:21:11  BTS[32] bts_cl_query_end: exit  (status = 0)
09:21:11  BTS[32] bts_lock_release: entry (name = '/ashworth/vessels_bts/1048885')
09:21:11  BTS[32] bts_lock_release: exit  (status = 0)
09:21:11  BTS[32] bts_fini: entry (errcode = 0)
09:21:11  BTS[32] bts_cl_fini: entry (bts_cl_init_value = 1)
09:21:11  BTS[32] bts_cl_init_clear: entry
09:21:11  BTS[32] bts_cl_init_clear: exit  (status = 0)
09:21:11  BTS[32] bts_cl_fini: exit  (bts_cl_init_value = 0, status = 0)
09:21:11  BTS[32] bts_lock_release: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: exit  (lock name: 'BTS_LOCK-EVP', status = 0)
09:21:11  BTS[32] bts_lock_release: exit  (status = 0)
09:21:11  BTS[32] bts_fini: exit  (status = 0)
09:21:11  BTS[32] bts_am_endscan: exit  (status = 0)
09:21:11  BTS[32] bts_am_close: entry
09:21:11  BTS[32] bts_init: entry
09:21:11  BTS[32] bts_lock_try: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: exit  (lock name: 'BTS_LOCK-EVP', status = 0)
09:21:11  BTS[32] bts_lock_try: exit  (status = 0)
```

```
09:21:11  BTS[32] bts_cl_init: entry (bts_cl_init_value = 0)
09:21:11  BTS[32] bts_cl_init_restore: entry
09:21:11  BTS[32] bts_cl_init_restore: exit  (status = 0)
09:21:11  BTS[32] bts_cl_init: exit  (bts_cl_init_value = 1, status = 0)
09:21:11  BTS[32] bts_gls_init: entry
09:21:11  BTS[32] bts_gls_init: exit  (status = 0)
09:21:11  BTS[32] bts_evp_check: entry
09:21:11  BTS[32] bts_evp_check: exit  (status = 0)
09:21:11  BTS[32] bts_auto_trace: (skipped)
09:21:11  BTS[32] bts_init: exit  (status = 0)
09:21:11  BTS[32] bts_am_spacename: entry
09:21:11  BTS[32] bts_am_spacename: exit  (spacename = 'bts_sbspace1', status = 0)
09:21:11  BTS[32] bts_am_userdata: (target = '/ashworth/vessels_bts/1048885')
09:21:11  BTS[32] bts_am_userdata_free: entry
09:21:11  BTS[32] bts_fini: entry (errcode = 0)
09:21:11  BTS[32] bts_cl_fini: entry (bts_cl_init_value = 1)
09:21:11  BTS[32] bts_cl_init_clear: entry
09:21:11  BTS[32] bts_cl_init_clear: exit  (status = 0)
09:21:11  BTS[32] bts_cl_fini: exit  (bts_cl_init_value = 0, status = 0)
09:21:11  BTS[32] bts_lock_release: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: entry (name = 'EVP')
09:21:11  BTS[32] bts_lock_name: exit  (lock name: 'BTS_LOCK-EVP', status = 0)
09:21:11  BTS[32] bts_lock_release: exit  (status = 0)
09:21:11  BTS[32] bts_fini: exit  (status = 0)
09:21:11  BTS[32] bts_am_close: exit  (status = 0)
09:21:11  BTS[32] bts_xact_end_xact: entry
09:21:11  BTS[32] bts_xact_bxh_init: entry
09:21:11  BTS[32] bts_xact_bxh_init: (XACT: named_memory(BTS_XACT_20))
09:21:11  BTS[32] bts_xact_bxh_init: (XACT: mi_named_get(BTS_XACT_20) failed: 2)
09:21:11  BTS[32] bts_xact_bxh_init: (XACT: mi_named_get(BTS_XACT_20) failure ignored)
09:21:11  BTS[32] bts_xact_bxh_init: exit  (status = 0, bxh = 0x00000000)
09:21:11  BTS[32] bts_xact_end_xact: exit  (status = 0, state = -1)
09:21:11  FSE Entry bts_inFseXactCallback end_xact
09:21:11  FSE Exit  bts_inFseXactCallback end_xact
09:21:11  BTS[32] bts_xact_post_xact: entry
09:21:11  BTS[32] bts_xact_bxh_init: entry
09:21:11  BTS[32] bts_xact_bxh_init: (XACT: named_memory(BTS_XACT_20))
09:21:11  BTS[32] bts_xact_bxh_init: (XACT: mi_named_get(BTS_XACT_20) failed: 2)
09:21:11  BTS[32] bts_xact_bxh_init: (XACT: mi_named_get(BTS_XACT_20) failure ignored)
09:21:11  BTS[32] bts_xact_bxh_init: exit  (status = 0, bxh = 0x00000000)
09:21:11  BTS[32] bts_xact_post_xact: exit  (status = 0, state = -1)
09:21:11  FSE Entry bts_inFseXactCallback post_xact
09:21:11  FSE Exit  bts_inFseXactCallback post_xact
```

Basic text search performance

The performance of basic text search queries depends on the **bts** index, disk space, configuration parameters, and BTS virtual processors.

When you create a **bts** index, include the following index parameters to improve the performance of basic test search queries:

- Set the delete index parameter to remove index information for deleted documents and release disk space. You can optimize the index manually or automatically after every delete operation.
- Set the query_default_field=* index parameter to create a composite index. When you run a basic text search query, the text from all the columns is searched in the `contents` field as if it was one string. This method can result in better query performance than using a UNION clause to combine the results of multiple queries on multiple **bts** indexes.

Include the following index parameters to improve the performance of building the **bts** index:

- Set the tempspace index parameter to the name of a temporary sbspace. Building the index in a temporary sbspace is faster than building the index in an sbspace that logs transactions.
- Set the xact_ramdirectory=yes index parameter to build the **bts** index in memory. Building the index in memory can be faster than building the index in a temporary sbspace.

The **bts** index is in an sbspace. If you defined multiple BTS virtual processors, each one can simultaneously process a different transaction. However, when a transaction that contains INSERT, DELETE, or UPDATE statements that affect the **bts** index is being committed, the transaction acquires an exclusive lock on the **bts** index. Any other concurrent transaction waits for up to 15 minutes for the lock to be released.

The **bts** index works in READ COMMITTED isolation level regardless of the isolation level that is set in the database server. The READ COMMITTED isolation level provides access only to rows that are committed. Uncommitted rows from other concurrent transactions are not accessible.

Disk space for the bts index

The size of the external **bts** index depends on the number of documents being indexed as well as the number of words and the number of unique words in those documents.

If you receive an I/O error such as `(BTSA1) - bts clucene error: IO error: File IO Write error`, check the online log. The probable cause is insufficient disk space. If this happens, drop the **bts** index with a DROP INDEX statement and recreate it on a disk with enough disk space.

To prevent running out of space for the **bts** index, create a dedicated sbspace for the **bts** index and a separate sbspace for temporary data. A separate sbspace for temporary data might also improve the speed of creating and updating the **bts** index.

See Preparing for basic text searching on page 118 for the procedure to create a **bts** index. See the *HCL OneDB™ Guide to SQL: Syntax* for instructions for the DROP INDEX statement.

Adding BTS virtual processors to run multiple queries simultaneously

You can increase the number of basic text search queries or other index operations that can run at the same time by adding additional BTS virtual processors.

About this task

Each Basic Text Search function, including bts_contains(), runs in a BTS virtual processor without yielding. If basic text search queries are slow because multiple users are running queries at the same time, you can add more BTS virtual processors so that queries run simultaneously, each in their own virtual processor.

To dynamically add BTS virtual processors for the current database server session:

Run the onmode -p command, specifying the number of virtual processors to add and the BTS virtual processor class. For example, the following command adds three BTS virtual processors: `onmode -p 3 bts`

Example

Alternatively, you can use the SQL administration API task() or admin() function with the **onmode** and **p** arguments to add BTS virtual processors.

To permanently increase the number of BTS virtual processors, set the value of the **VPCLASS bts** configuration parameter in the `onconfig` file and then restart the database server. If the `onconfig` file contains an existing entry for the **VPCLASS bts** configuration parameter, update that entry; otherwise, add a new entry for the **VPCLASS bts** configuration parameter.

For more information about the onmode utility or the SQL administration API, see the *HCL OneDB™ Administrator's Reference*,

Tune configuration parameters for basic text searching

You can optimize the performance of basic text searches by tuning certain configuration parameters.

AUTO_READAHEAD

The AUTO_READAHEAD configuration parameter enables automatic read-ahead. Pages that are brought into the bufferpool cache during sequential scans of data records can improve the performance of a query when the server detects that the query is encountering I/O.

BUFFERPOOL

The BUFFERPOOL configuration parameter defines a buffer pool for pages that correspond to each unique page size that is used by your dbspaces. You can specify information about the buffer pool, including its size, the number of LRU queues in the buffer pool, the number of buffers in the buffer pool, and minimum and maximum percentages of modified pages in the LRU queues.

RESIDENT

The RESIDENT configuration parameter specifies whether the resident portion of shared memory remains resident in operating system physical memory. If your operating system supports forced residency, you can improve the performance of searches by specifying that the resident portion of shared memory is not swapped to disk. Set the RESIDENT configuration parameter to 1 (on).

VPCLASS

You can add the noage option to the VPCLASS configuration parameter setting for the BTS virtual processor. The noage option improves performance by disabling priority aging by the operating system. The BTS virtual processor that is created automatically does not include the noage option.

PRELOAD_DLL_FILE

You can add the BTS shared library, `bts.bld`, to a PRELOAD_DLL_FILE configuration parameter setting to preload the BTS shared library when the server starts instead of loading the library the first time you run a BTS function or create a **bts** index. For example, add the following line to your `onconfig` file:

```
PRELOAD_DLL_FILE $ONEDB_HOME/extend/bts.version/bts.bld
```

The *version* is the specific version number for the extension. Run the bts_release() function to find the correct version number. The version number of the BTS extension can change in any fix pack or release. After you upgrade, you must update the value of the PRELOAD_DLL_FILE configuration parameter if the version number of the BTS extension changed.

Important: Do not preload the `bts_cl.bld` file or the BTS extension does not operate properly.

Basic text search error codes

Basic text searching has specific error messages.

The following table lists error codes for basic text searching.

SQLstate	Description
BTS01	bts error, assertion failed. File %FILE%, line %LINE%
BTS02	bts internal error. File %FILE%, line %LINE%
BTS03	bts error - could not set trace level to %PARAM1% for trace class %PARAM2%
BTS04	bts error - could not set trace output file to %PARAM1%
BTS05	bts error - unique index not supported
BTS06	bts error - cluster index not supported
BTS08	bts error - cannot query the table %TABLENAME%
BTS09	bts error - BTS index only supports extspaces and sbspaces
BTS10	bts error - cannot get connection descriptor
BTS11	bts error - extspace not specified
BTS12	bts error - cannot determine index owner
BTS13	bts error - cannot determine index name
BTS14	bts error - cannot create directory %PARAM1%
BTS15	bts error - current vpclass (%VPCLASS%) is not specified as noyield
BTS16	bts error - too many virtual processors running (%NUMVPS%) for the current vpclass (%VPCLASS%), 1 is the maximum
BTS17	bts error - out of memory
BTS18	bts error - SQL Boolean expression are not supported with bts_contains
BTS19	bts error - cannot query with a null value
BTS20	bts error - invalid value for index delete parameter: %PARAM1% should be either immediate or deferred

SQLstate	Description
BTS21	bts error - unsupported type: %PARAM1%
BTS22	bts error - bts_contains requires an index on the search column
BTS23	bts error - cannot register end-of-transaction-callback
BTS24	bts error - invalid value for %PARAM1% parameter: %PARAM2% should be an integer value greater than 0
BTS25	bts error - CLOB or BLOB is too large, must be less than or equal to 2,147,483,647 bytes
BTS26	bts error - clob or blob is too large, must be less than or equal to 2,147,483,647
BTS27	bts error - BTS indexes in external spaces only permitted on primary or standard servers
BTS28	bts error - invalid value for the %PARAM1% parameter: %PARAM2% should be "unlimited" or an integer value greater than 0
BTS29	bts error - invalid value for the %PARAM1% parameter: %PARAM2% should be either "and" or "or"
BTS30	bts error - invalid value for the PARAM1% parameter: %PARAM2% should be either yes or no
BTS31	bts error - invalid value for the %PARAM1% parameter: %PARAM2% should be either yes, yes_with_tag or no
BTS32	bts error - invalid value for the %PARAM1% parameter: %PARAM2% should be either yes, yes_with_database_name or no
BTS33	bts error - incorrect value for the %PARAM1% parameter: %PARAM2% should be either yes, yes_with_positions, yes_with_offsets or no
BTS34	bts error - uppercase characters are not allowed in stopwords
BTS35	bts internal error - mi_open() failed. File %FILE%, line %LINE%
BTS36	bts internal error - mi_lo_open() failed. File %FILE%, line %LINE%
BTS37	bts internal error - mi_lo_seek() failed. File %FILE%, line %LINE%
BTS38	bts internal error - mi_lo_read() failed. File %FILE%, line %LINE%
BTS39	bts internal error - ifx_int8toasc() failed. File %FILE%, line %LINE%
BTS40	bts internal error - mi_lo_spec_init() failed. File %FILE%, line %LINE%
BTS41	bts internal error - mi_lo_create() failed. File %FILE%, line %LINE%
BTS42	bts internal error - mi_lo_increfcount() failed. File %FILE%, line %LINE%
BTS43	bts internal error - ifx_int8cvlong() failed. File %FILE%, line %LINE%
BTS44	bts internal error - mi_lo_write() failed. File %FILE%, line %LINE%
BTS45	bts error - cannot open file %FILENAME%
BTS46	bts error - cannot create file %FILENAME%

SQLstate	Description
BTS47	bts error - xml syntax error
BTS48	bts error - invalid hex specification: \x%PARAM1%%PARAM2%
BTS49	bts error - the GLS character name '%PARAM1%' is not found
BTS50	bts error - if either xmltags is specified or all_xmltags is enabled, then include_contents must be enabled if strip_xmltags is enabled
BTS51	bts error - xmlpath_processing cannot be enabled unless either xmltags is specified or all_xmltags is enabled.
BTS52	bts error – parameter %PARAM1% and %PARAM2% parameters are mutually exclusive
BTS53	bts error - invalid value for the %PARAM1% parameter: %PARAM2% should be a lower value
BTS54	bts error - cannot write to file %FILENAME%
BTS55	bts error - cannot read from file %FILENAME%
BTS56	bts error - bad magic number on file %FILENAME%
BTS57	bts error - the specified table (%TABLENAME%) is not in the database
BTS58	bts error - column (%COLUMNNAME%) not found in specified table (%TABLENAME%)
BTS59	bts error - column (%COLUMNNAME%) in specified table (%TABLENAME%) is not of type char, varchar, nchar, nvarchar or lvarchar
BTS60	bts error - cannot acquire exclusive lock for %PARAM1%
BTS61	bts error - cannot acquire read lock for %PARAM1%
BTS62	bts error - cannot acquire write lock for %PARAM1%
BTS63	bts error - parameter %PARAM1% is not implemented yet"
BTS64	bts error - %PARAM1% contains a '/' character which indicates an xmlpath however xmlpath_processing is not enabled. Either remove the '/' in the xmltag or enable xmlpath_processing"
BTS65	bts error - invalid value for tempspace parameter: %PARAM1% should be an existing extspace or sbspace
BTS66	bts error - the include_contents cannot be enabled unless the xmltags, all_xmltags, json_names or all_json_names parameter is enabledbts error - include_contents cannot be enabled unless either xmltags is specified or all_xmltags is enabled
BTS67	bts error - include_namespaces cannot be enabled unless either xmltags is specified or all_xmltags is enabled
BTS68	bts error - include_subtag_text cannot be enabled unless either xmltags is specified or all_xmltags is enabled
BTS69	bts error - %PARAM1% only works with on one bts virtual processor
BTS70	bts internal error - mi_lo_specset_sbspace() failed. File %FILE%, line %LINE%
BTS71	bts internal error - mi_lo_stat() failed. File %FILE%, line %LINE%

SQLstate	Description
BTS72	bts internal error - mi_lo_stat_cspec() failed. File %FILE%, line %LINE%
BTS73	bts error - sbspace %PARAM1% is not logged
BTS74	bts error - sbspace for FSE is not set
BTS75	bts error - SBSPACENAME not set in onconfig file
BTS76	bts error - transaction uses too much memory. Perform smaller transactions or increase the value of the xact_memory parameter on the index
BTS77	bts error - invalid value for xact_memory: %PARAM1% should be either unlimited or the maximum amount of memory (between 1 and %PARAM2% kilobytes)
BTS78	bts error - SQL create index and drop index are not supported on updatable secondary nodes
BTS79	bts error - not implemented yet
BTS80	bts error - database must be logged
BTS81	bts error - not in a transaction
BTS82	bts error - xpath syntax error
BTS83	bts internal error - mi_file_seek failed. File %FILE%, line %LINE%
BTS84	bts internal error - mi_lo_decrefcount failed. File %FILE%, line %LINE%
BTS85	bts internal error - mi_lo_from_string failed. File %FILE%, line %LINE%
BTS86	bts internal error - mi_lo_release() failed. File %FILE%, line %LINE%
BTS87	bts internal error - mi_lo_to_string failed. File %FILE%, line %LINE%
BTS88	bts error - no lo handle found in file %PARAM1%
BTS89	bts error - valid lo handle found in file %PARAM1%
BTS90	bts error - CLucene index exists and is locked
BTS91	bts error - CLucene index exists
BTS92	bts error - CLucene index does not exist
BTS93	bts error - the parameter %PARAM1% should be in the form of name="value"
BTS94	bts error - missing a double quotation mark: ". The parameter %PARAM1% should be in the form of name="value"
BTS95	bts error - missing the closing parenthesis:). The parameter %PARAM1% should be in the form of name="(values)"
BTS96	bts error - missing a double quotation mark: ". The parameter %PARAM1% should be in the form of name="(values)"

SQLstate	Description
BTS97	bts error - missing a comma (,) between parameters
BTS98	bts error - duplicate parameters, %PARAM1%, were specified
BTS99	bts clucene error: Unknown error: %PARAM1%
BTSA1	bts clucene error: IO error: %PARAM1%
BTSA2	bts clucene error: Null pointer error: %PARAM1%
BTSA3	bts clucene error: Runtime error: %PARAM1%
BTSA4	bts clucene error: Illegal argument: %PARAM1%
BTSA5	bts clucene error: Parse error: %PARAM1%
BTSA6	bts clucene error: Token manager error: %PARAM1%
BTSA7	bts clucene error: Unsupported operation: %PARAM1%
BTSA8	bts clucene error: Invalid state: %PARAM1%
BTSA9	bts clucene error: Index out of bounds: %PARAM1%
BTSB0	bts clucene error: Too Many Clauses: %PARAM1%
BTSB1	bts clucene error: RAM Transaction error: %PARAM1%
BTSB2	bts clucene error: Invalid Cast: %PARAM1%
BTSC0	GLS Error: An attempt to create a locale with incompatible code sets has occurred
BTSC1	GLS Error: Bad format found in the codeset registry file
BTSC2	GLS Error: Either locale or code set conversion specifiers, i.e., GLS or NLS environment variables, is incorrect, or the codeset name registry file could not be found
BTSC3	GLS Error: Not enough memory to allocate a new locale object or a new codeset conversion object
BTSC4	GLS Error: The locale contains characters that are wider than the library allows
BTSC5	GLS Error: The locale object version is not compatible with the current library
BTSC6	GLS Error: The locale or codeset conversion file could not be found, is not readable, or has the wrong format
BTSC7	GLS Error: Unknown %PARAM1%
BTSC8	bts internal error - mi_lo_stat_size() failed. File %FILE%, line %LINE%
BTSC9	bts internal error - biginttoasc() failed. File %FILE%, line %LINE%
BTSD0	bts error - invalid canonical map[%PARAM1%]: zero length original character string
BTSD1	bts error - invalid canonical map[%PARAM1%]: %PARAM2% is an uppercase character. Uppercase characters are not allowed in canonical maps

SQLstate	Description
BTSD2	bts error - invalid canonical map[%PARAM1%]: missing %PARAM2% in mapped characters specification
BTSD3	bts error - invalid canonical map[%PARAM1%]: missing %PARAM2% in original characters specification
BTSD4	bts error - invalid canonical map[%PARAM1%]: missing : in mapped characters specification
BTSD5	bts error - invalid canonical map[%PARAM1%]: missing] in alternates of original characters specification
BTSD6	bts error - invalid canonical map[%PARAM1%]: spaces found in original character string at %PARAM2%
BTSD7	bts error - invalid canonical map[%PARAM1%]: trailing characters found
BTSD8	bts error - missing the closing parenthesis,), in a string that has an opening parenthesis: (
BTSD9	bts error - missing the column name in table:%PARAM1%. Use the form table:table_name.column_name
BTSE0	bts error - parameter %PARAM1% is not updatable
BTSE1	bts error - unknown parameter name: %PARAM1%
BTSE2	bts error - recursive params parameter
BTSE3	bts error - invalid value for the %PARAM1% parameter: %PARAM2% is too long
BTSE4	bts error - invalid flag for the create_mode parameter: %PARAM1%
BTSE5	bts error - invalid value for the create_mode parameter: %PARAM1% should be a hexadecimal number
BTSE6	bts error - %PARAM1% encoding is not supported for %PARAM2%
BTSE7	bts error - UDR analyzer function %PARAM1% not found
BTSE8	bts error - UDR analyzer function id not found for %PARAM1%
BTSE9	bts error - default analyzer already set
BTSF0	bts error - empty stopwords field specification
BTSF1	bts error - invalid analyzer value: %PARAM1%
BTSF2	bts error - invalid value for the analyzer parameter: %PARAM1%
BTSF3	bts error - no analyzer specified for field: %PARAM1%
BTSF4	bts error - no field name in field:analyzer specification: %PARAM1%
BTSF5	bts error - the field %PARAM1% appears multiple times in the stopwords list
BTSF6	bts error - a stopwords list cannot be specified for the analyzer: %PARAM1%
BTSF7	bts error - too many colons found in stopwords field specification
BTSF8	bts error - too many colons found in field:analyzer specification: %PARAM1%
BTSF9	bts error - there is no snowball stemmer language specified after the period

SQLstate	Description
BTSG0	bts error - there is no snowball stemmer language support for the $DB_LOCALE setting: %PARAM1%
BTSG1	bts error - there is no snowball stemmer language support for the specified language: %PARAM1%
BTSG2	bts error - internal index length %PARAM1% is too long. The maximum is %PARAM2%
BTSG3	bts error - bts_lock_setup: cannot get vp lock pointer
BTSG4	bts error - bts_lock_setup: vp is not locked
BTSG5	bts error - bts_lock_setup: vp is not locked by the current transaction
BTSG6	bts error - not (-) operator may not be specified in thesaurus
BTSG7	bts error - and (+) operator may not be specified in thesaurus
BTSG8	bts error - cannot determine index owner of thesaurus index %PARAM1%
BTSG9	bts error - cannot lock thesaurus index %PARAM1%
BTSH0	bts error - cannot read thesaurus index parameters for %PARAM1%
BTSH1	bts error - the index %PARAM1% does not have the thesaurus parameter set
BTSH2	bts error - thesaurus index cannot be fragmented
BTSH3	bts error - invalid term found in thesaurus. Only word terms should be specified
BTSH4	bts error - the %PARAM1% attribute must be specified
BTSH5	bts error - the text or file attribute must be specified
BTSH6	bts error - the copy_temp attribute can only be specified on an index in an sbspace
BTSH7	bts error - the field is not in the document
BTSH8	bts error - the directory cannot contain a bts index
BTSH9	bts error - the ID is out of bounds
BTSI0	bts error - must be a DBSA to use parameter %PARM1%
BTSI1	bts error - cannot cast json value to bson value: %MSG%
BTSI2	bts error - bson or json types cannot be used in a composite index
BTSI3	bts error - the json_path_processing parameter cannot be enabled unless the json_names parameter is specified or the all_json_names parameter is enabled
BTSI4	bts error - the only_json_values parameter requires that the include_contents is enabled when json_names is specified or all_json_names is enabled
BTSI5	bts error - bson format error decoding type %TYPE% (%SIZE% bytes) at byte %POS% of %MAX% total bytes

SQLstate	Description
BTSI6	bts error - the total number of expected bytes recorded in the bson value, %LEN%, exceeds the actual length of the bson value, %MAX% bytes
BTSI7	bts error - bson format error: %BYTES% bytes of the bson value after the end-of-value mark were not processed
BTSI8	bts error - the json_names, all_json_names, only_json_values, or ignore_json_format_errors parameter can be specified only with bson or json types
BTSI9	bts error - bson or json types cannot be used when xmltags is specified or all_xmltags is enabled
BTSJ0	bts error - json format error at %POS%: extra characters following right brace (})
BTSJ1	bts error - json format error at %POS%: missing right square bracket (])
BTSJ2	bts error - json format error at %POS%: invalid number
BTSJ3	bts error - json format error at %POS%: missing colon (:) separator between name and value
BTSJ4	bts error - json format error at %POS%: missing document
BTSJ5	bts error - json format error at %POS%: missing left brace ({)
BTSJ6	bts error - json format error at %POS%: missing name
BTSJ7	bts error - json format error at %POS%: missing right brace (})
BTSJ8	bts error - json format error at %POS%: missing value
BTSJ9	bts error - json format error at %POS%: missing double quote (") in string

Hierarchical data type

The node data type helps to resolve a difficult relational database problem–transitive closure.

This transitive closure problem is endemic to data management problems, and not particularly well addressed by the relational model. The same basic problem is found modeling organizational hierarchies, networks, manufacturing and process control databases.

You can use the node data type to improve query performance for many recursive queries. Using the node data type can also ease the burden of transitive dependency in the relational database model. *Transitive dependency* occurs when a non-key attribute is dependent on another non-key attribute. This relationship frequently has multiple levels of attribute dependency. The problem usually is seen when you model organizational hierarchies, networks, and databases for manufacturing and process control.

The node data type for querying hierarchical data

The node data type is an opaque type of variable length up to 256 characters.

The Scheduler must be running in the database server. If the Scheduler is not running when you create a node data type, a message that the data type is not found is returned.

The database that contains the node data types must meet the following requirements:

- The database must be logged.
- The database must not be defined as an ANSI database.

If you attempt to create a node data type in an unlogged or ANSI database, the message `DataBlade registration failed` is printed in the database server message log.

Operations involving Enterprise Replication are supported.

Deep copy and LIKE matching statements are not supported.

Troubleshooting the node data type

Error message specific to the node data type have the prefix UND. You can enable tracing on the node data type to diagnose problems.

You might receive the following errors:

Error	Description
UNDE1: Invalid input string.	A node is invalid. Nodes cannot end in 0.
UNDE2: Illegal character found in input string.	An argument contains an illegal character. Nodes can contain only numeric characters.
UNDE3: Third input parameter is not descendant of first input parameter.	The third argument of a Graft function is not a descendant of the first argument.
UNDE4: Index to node element should be greater than or equal to 1.	A problem exists with the node indexing.

To enable tracing, create a trace class by inserting a record into the **systemtraceclasses** system catalog:

```
INSERT INTO informix.systraceclasses(name) VALUES ('Node');
```

For more details regarding tracing, see the *HCL OneDB™ Guide to SQL: Reference*.

Node data type functions

Use these functions in queries involving the node data type.

Ancestors() function

The Ancestors() function is an iterator function that returns ancestor nodes. The Ancestors function recursively calls itself with the output from IsAncestor.

Syntax

```
Ancestors(node)
```

node

The node for which you want to find all ancestor nodes.

Example

```
EXECUTE FUNCTION ancestors('1.2.3.4.5.6.7.8.9');
```

This function returns the following eight rows as ancestor nodes:

```
1.2.3.4.5.6.7.8
1.2.3.4.5.6.7
1.2.3.4.5.6
1.2.3.4.5
1.2.3.4
1.2.3
1.2
1.0
```

Compare() function

The Compare( ) function compares two node types to determine if they are the same.

Returns: -1, 0, or 1.

-1

The first argument is less than the second.

0

The arguments are equal.

1

The first argument is greater than the second.

Syntax

```
compare(node1, node2)
```

node1

The first node to compare.

node2

The node to which the first argument is compared

Example

```
CREATE TABLE nodetab1 (col1 node);
INSERT INTO nodetab1 VALUES ('1.0');
INSERT INTO nodetab1 VALUES ('2.0');

SELECT n1.col1, n2.col1, Compare (n1.col1, n2.col1)
```

```
FROM nodetab1 n1, nodetab1 n2;

col1          1.0
col1          1.0
(expression)  0

col1          2.0
col1          1.0
(expression)  1

col1          1.0
col1          2.0
(expression)  -1
```

Depth() function

The Depth( ) function returns the number of levels in the specified node.

Returns: integer

Syntax

```
Depth(node)
```

node

The node for which you want to determine depth.

Examples

Example 1

```
EXECUTE FUNCTION DEPTH('1.22.3');
```

Returns: 3

Example 2

```
EXECUTE FUNCTION DEPTH('6.5.4.3.2.1');
```

Returns: 6

GetMember() function

The GetMember( ) function returns information about a node level, returns integer. The GetMember() function returns specific parts of the node argument. The second argument specifies the level you want returned. A NULL is returned if no corresponding level exists.

Returns: integer or NULL

Syntax

```
GetMember(node, integer)
```

node

integer

Example

```
CREATE TABLE nodetab1 (col1 node);
INSERT INTO nodetab1 VALUES ('1.0');
INSERT INTO nodetab1 VALUES ('1.1.1');
INSERT INTO nodetab1 VALUES ('1.1.2');
INSERT INTO nodetab1 VALUES ('1.1.2.1');
INSERT INTO nodetab1 VALUES ('2.0');

SELECT  col1, GetMember(col1, 3)
FROM    nodetab1;

col1          1.0
(expression)

col1          1.1.1
(expression)  1

col1          1.1.2
(expression)  2

col1          1.1.2.1
(expression)  2

col1          2.0
(expression)
```

GetParent() function

The GetParent( ) function returns the parent of a node. If the node does not have a parent NULL is returned.

Returns: node or NULL

Syntax

```
GetParent(node)
```

node

The child node whose parent you want to determine.

Example

```
CREATE TABLE nodetab1 (col1 node);
INSERT INTO nodetab1 VALUES ('1.0');
INSERT INTO nodetab1 VALUES ('1.1.1');
INSERT INTO nodetab1 VALUES ('1.1.2');
INSERT INTO nodetab1 VALUES ('1.1.2.1');
INSERT INTO nodetab1 VALUES ('2.0');

SELECT  col1, GetParent(col1)
FROM    nodetab1;

col1          1.0
```

```
(expression)

col1          1.1.1
(expression)  1.1

col1          1.1.2
(expression)  1.1

col1          1.1.2.1
(expression)  1.1.2

col1          2.0
(expression)
```

GreaterThanOrEqual() function

The GreaterThanOrEqual() function compares two nodes to determine if the first is greater or equal to the second. Implements the comparison operator and can be used in SQL statements either using the function name or the corresponding symbol.

Returns: Boolean

Syntax

```
GreaterThanOrEqual(node1, node2)
```

node1

The node that you are will compare against.

node2

The node that you are checking to see if it is greater than or equal to *node1*.

Examples

Example 1

```
SELECT *
FROM tablename
WHERE GreaterThanOrEqual(nodecolumn, “1.4");
```

Example 2

```
SELECT *
FROM tablename
WHERE nodecolumn >= “1.4";
```

This example is the same as Example 1, except a greater than or equal sign is used in place of the function name.

Increment() function

The Increment( ) function determines the next node at the same level. You can also increase the level of a node by one at a specified level.

Returns: node

Syntax

```
Increment(node, integer)
```

node

The starting node to increment from.

integer

The node member to increment. If you do not specify this argument, the next node at the same level as *node1* is returned.

Examples

Example 1

```
EXECUTE FUNCTION Increment('1.2.3');
(expression)  1.2.4
```

This example uses only one argument. The result shows the next node at the same level.

Example 2

```
EXECUTE FUNCTION Increment('1.2.3', 3);
(expression)  1.2.4
```

This example increments the member in position three, whose value is 3.

Example 3

```
EXECUTE FUNCTION Increment('1.2.3', 1);
(expression)  2.0
```

This example increments the first node member.

IsAncestor() function

The IsAncestor() function lets you determine if a specific node is an ancestor of another. This function is the opposite of IsDescendant().

Returns: Boolean

Syntax

```
IsAncestornode1, node2)
```

node1

The parent node for which you want to find an ancestor.

node2

The node that you want to determine whether it is an ancestor of `node1`.

Examples

Example 1

```
CREATE TABLE nodetab1 (col1 node);
INSERT INTO nodetab1 VALUES ('1.0');
INSERT INTO nodetab1 VALUES ('1.1');
INSERT INTO nodetab1 VALUES ('1.1.1');

SELECT  n1.col1, n2.col1, IsAncestor (n1.col1, n2.col1)
FROM    nodetab1 n1, nodetab1 n2;

col1          1.0
col1          1.1
(expression)  t

col1          1.0
col1          1.1.1
(expression)  t

col1          1.1
col1          1.1.1
(expression)  t

col1          1.1.1
col1          1.1
(expression)  f
```

Example 2

```
SELECT  col1
FROM    nodetab1 n1
WHERE   isAncestor(col1, '1.1.2');

col1  1.0

col1  1.1
```

IsChild() function

The IsChild( ) function determines whether a node is a child of another node. This is the opposite of IsParent().

Returns: Boolean

Syntax

```
IsChild(node1, node2)
```

node1

The node that you want to determine whether it is a child of *node2*.

node2

The parent node for which you want to find a child.

Example

```
CREATE TABLE nodetab1 (col1 node);
INSERT INTO nodetab1 VALUES ('1.0');
INSERT INTO nodetab1 VALUES ('1.1');
INSERT INTO nodetab1 VALUES ('1.1.1');

SELECT  n1.col1, n2.col1, IsChild (n1.col1, n2.col1)
FROM    nodetab1 n1, nodetab1 n2;

col1          1.1
col1          1.0
(expression)  t

col1          1.1.1
col1          1.0
(expression)  f

col1          1.0
col1          1.1
(expression)  f

col1          1.1
col1          1.1
(expression)  f

col1          1.1.1
col1          1.1
(expression)  t

col1          1.0
col1          1.1.1
(expression)  f
```

IsDescendant() function

The IsDescendant( ) function lets you determine if a specific node is a descendant of another. This function is the opposite of IsAncestor().

Returns: Boolean

Syntax

```
IsDescendant(node1, node2)
```

node1

The node that you want to determine whether it is a descendant of `node1`.

node2

The parent node for which you want to find a descendant.

Example

```
CREATE TABLE nodetab1 (col1 node);
INSERT INTO nodetab1 VALUES ('1.0');
```

```
INSERT INTO nodetab1 VALUES ('1.1');
INSERT INTO nodetab1 VALUES ('1.1.1');

SELECT  n1.col1, n2.col1, IsDescendant (n1.col1, n2.col1)
FROM    nodetab1 n1, nodetab1 n2;

col1          1.0
col1          1.0
(expression)  f

col1          1.1
col1          1.0
(expression)  t

col1          1.1.1
col1          1.0
(expression)  t

col1          1.0
col1          1.1
(expression)  f
```

IsParent() function

The IsParent( ) function lets you determine if a specific node is a parent of another. This function is the opposite of IsChild().

Returns: Boolean

Syntax

```
IsParent(node1, node2)
```

node1

The node that you want to determine whether it is a parent of *node2*.

node2

The descendant node for which you want to find a parent.

Example

```
CREATE TABLE nodetab1 (col1 node);
INSERT INTO nodetab1 VALUES ('1.0');
INSERT INTO nodetab1 VALUES ('1.1');
INSERT INTO nodetab1 VALUES ('1.1.1');

SELECT  n1.col1, n2.col1, IsParent (n1.col1, n2.col1)
FROM    nodetab1 n1, nodetab1 n2;

col1          1.0
col1          1.1
(expression)  t

col1          1.1
col1          1.1.1
(expression)  t
```

```
col1          1.0
col1          1.1.1
(expression)  f
```

Length() Node function

The Length( ) function returns the number of levels in the specified node and is equivalent to the Depth() function. This is the name of the function that was included in Node Version 1.0 and supported for continuity.

Returns: integer

Syntax

```
Length(node::node)
```

node

The node for which you want to determine depth, which is how many levels are in the node.

Example

```
execute function length('1.22.3'::node);
(expression)  3
```

LessThan() function

The LessThan( ) function compares two nodes to determine which is less. Implements the comparison operator and can be used in SQL statements either using the function name or the corresponding symbol.

Returns: Boolean

Syntax

```
LessThan(node1, node2)
```

node1

The node that you are will compare against.

node2

The node that you are checking to see if it is less than *node1*.

Examples

Example 1

```
SELECT * FROM tablename WHERE LessThan(nodecolumn, '1.4');
```

The following list includes nodes that are less than 1.4:

1. 0.4
2. 1.3

3. 1.3.66
4. 1.1.1.1

The following list includes nodes that are greater than 1.4:

1. 1.4.1.1
2. 1.5
3. 2.0

Example 2

```
SELECT * FROM tablename WHERE nodecolumn < '1.4';
```

LessThanOrEqual() function

The LessThanOrEqual() function compares two nodes to determine if the first is less or equal to the second. Implements the comparison operator and can be used in SQL statements either using the function name or the corresponding symbol.

Returns: Boolean

Syntax

```
LessThanOrEqual(node1, node2)
```

node1

The node that you are will compare against.

node2

The node that you are checking to see if it is less than or equal to *node1*.

Examples

Example 1

```
SELECT * FROM tablename
WHERE LessThanOrEqual(nodecolumn, '1.4');
```

This example searches the values in the node column of the table to find the node with the value 1.4.

Example 2

```
SELECT * FROM tablename
WHERE nodecolumn <= '1.4';
```

This example is the equivalent to the first, but uses symbols instead of the function name.

NewLevel() function

The NewLevel( ) function creates a new node level. This function simply returns a new node value under the argument node. This function is independent of table values. The function does not check for duplication.

Returns: node

Syntax

```
NewLevel(node)
```

node

The node under which a new node is created

Example

```
EXECUTE FUNCTION NewLevel ('1.2.3');
(expression)  1.2.3.1
```

NodeRelease() function

The NodeRelease( ) function reports the release and version information of the node data type. This function takes no arguments.

Returns: string

Syntax

```
NodeRelease()
```

node

NotEqual() function

The NotEqual( ) function compares two nodes to determine whether they are not equal. Implements the comparison operator and can be used in SQL statements either using the function name or the corresponding symbol. The opposite function is Equal().

Returns: Boolean

Syntax

```
NotEqual(node1, node2)
```

node1

The node against which you will test for inequality.

node2

The node that you will compare to the first to test for inequality.

Examples

Example 1

```
SELECT * FROM tablename WHERE NotEqual(nodecolumn, '1.4');
```

Example 2

```
SELECT * FROM tablename WHERE nodecolumn != '1.4';
```

This example is the same as Example 1, except a not equal sign is used in place of the function name.

SQL Packages Extension

The SQL packages extension provides SPL routines that you can use in an application that is compatible with database servers other than HCL OneDB™.

The SQL packages extension is a built-in extension in the `extend/excompat` directory of your installation. You must manually register the extension using the instructions in the *HCL OneDB™ DataBlade® Module Installation and Registration Guide*.

The database that contains the SQL packages extension must meet the following requirements or the extension is not registered:

- The database must be logged.
- The database must not be defined as an ANSI database.

The following modules are included in the SQL packages extension:

- DBMS_ALERT
- DBMS_LOB
- DBMS_OUTPUT
- DBMS_RANDOM
- UTL_FILE

DBMS_ALERT package

The DBMS_ALERT package provides a set of procedures for registering for alerts, sending alerts, and receiving alerts.

Alerts are stored in the DBMS_ALERT_EVENTS, DBMS_ALERT_REGISTERED, and DBMS_ALERT_SIGNALED tables which are created in your database when you register the package.

The DBMS_ALERT package includes the following system-defined routines.

Table 26. System-defined routines available in the DBMS_ALERT package

Routine name	Description
REGISTER procedure on page 215	Registers the current session to receive a specified alert.
REMOVE procedure on page 215	Removes registration for a specified alert.
REMOVEALL procedure on page 215	Removes registration for all alerts.

Table 26. System-defined routines available in the DBMS_ALERT package (continued)

Routine name	Description
SIGNAL procedure on page 216	Signals the occurrence of a specified alert.
SET_DEFAULTS procedure on page 216	Sets the polling interval for the WAITONE and WAITANY procedures.
WAITANY procedure on page 216	Waits for any registered alert to occur.
WAITONE procedure on page 217	Waits for a specified alert to occur.

Usage notes

The procedures in the DBMS_ALERT package are useful when you want to send an alert for a specific event. For example, you might want to send an alert when a trigger is activated as the result of changes to one or more tables.

REGISTER procedure

The REGISTER procedure registers the current session to receive a specified alert.

Syntax

```
DBMS_ALERT.REGISTER(name)
```

Procedure parameters

name

An input argument of type VARCHAR(128) that specifies the name of the alert.

REMOVE procedure

The REMOVE procedure removes registration from the current session for a specified alert.

Syntax

```
DBMS_ALERT.REMOVE(name)
```

Procedure parameters

name

An input argument of type VARCHAR(128) that specifies the name of the alert.

REMOVEALL procedure

The REMOVEALL procedure removes registration from the current session for all alerts.

Syntax

```
DBMS_ALERT.REMOVEALL
```

SET_DEFAULTS

The SET_DEFAULTS procedure sets the polling interval that is used by the WAITONE and WAITANY procedures.

Syntax

```
DBMS_ALERT.SET_DEFAULTS( sensitivity )
```

Procedure parameters

sensitivity

An input argument of type INTEGER that specifies an interval in seconds for the WAITONE and WAITANY procedures to check for signals. If a value is not specified, then the interval is 1 second by default.

SIGNAL procedure

The SIGNAL procedure signals the occurrence of a specified alert. The signal includes a message that is passed with the alert. The message is distributed to the listeners (processes that have registered for the alert) when the SIGNAL call is issued.

Syntax

```
DBMS_ALERT.SIGNAL(name,message)
```

Procedure parameters

name

An input argument of type VARCHAR(128) that specifies the name of the alert.

message

An input argument of type VARCHAR(32672) that specifies the information to pass with this alert. This message can be returned by the WAITANY or WAITONE procedures when an alert occurs.

WAITANY procedure

The WAITANY procedure waits for any registered alerts to occur.

Syntax

```
DBMS_ALERT.WAITANY(name,message,status,timeout)
```

Procedure parameters

name

An output argument of type VARCHAR(128) that contains the name of the alert.

message

An output argument of type VARCHAR(32672) that contains the message sent by the SIGNAL procedure.

status

An output argument of type INTEGER that contains the status code returned by the procedure. The following values are possible

0

An alert occurred.

1

A timeout occurred.

timeout

An input argument of type INTEGER that specifies the amount of time in seconds to wait for an alert.

WAITONE procedure

The WAITONE procedure waits for a specified alert to occur.

Syntax

```
DBMS_ALERT.WAITONE(name,message,status,timeout)
```

Procedure parameters

name

An input argument of type VARCHAR(128) that specifies the name of the alert.

message

An output argument of type VARCHAR(32672) that contains the message sent by the SIGNAL procedure.

status

An output argument of type INTEGER that contains the status code returned by the procedure. The following values are possible

0

An alert occurred.

1

A timeout occurred.

timeout

An input argument of type INTEGER that specifies the amount of time in seconds to wait for the specified alert.

DBMS_LOB package

The DBMS_LOB package provides the capability to operate on large objects.

In the following sections describing the individual procedures and functions, lengths and offsets are measured in bytes if the large objects are BLOBs. Lengths and offsets are measured in characters if the large objects are CLOBs.

The DBMS_LOB package supports LOB data up to 10M bytes.

The DBMS_LOB package includes the following routines.

Table 27. System-defined routines available in the DBMS_LOB package

Routine Name	Description
APPEND procedure on page 219	Appends one large object to another.
COMPARE function on page 219	Compares two large objects.
COPY procedure on page 220	Copies one large object to another.
ERASE procedure on page 221	Erases a large object.
GETLENGTH function on page 221	Gets the length of the large object.
INSTR function on page 222	Gets the position of the nth occurrence of a pattern in the large object starting at offset.
READ procedure on page 222	Reads a large object.
SUBSTR function on page 223	Gets part of a large object.
TRIM procedure on page 223	Trims a large object to the specified length.
WRITE procedure on page 224	Writes data to a large object.

In partitioned database environments, you will receive an error if you execute any of the following routines inside a WHERE clause of a SELECT statement:

- dbms_lob.compare
- dbms_lob.get_storage_limit
- dbms_lob.get_length
- dbms_lob.instr
- dbms_lob.isopen
- dbms_lob.substr

The following table lists the public variables available in the package.

Table 28. DBMS_LOB public variables

Public variables	Data type	Value
lob_readonly	INTEGER	0
lob_readwrite	INTEGER	1

APPEND procedures

The APPEND procedures provide the capability to append one large object to another.

Note: Both large objects must be of the same type.

Syntax

```
APPEND_BLOB(dest_lob, src_lob)
```

```
APPEND_CLOB(dest_lob, src_lob)
```

Parameters

dest_lob

An input or output argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator for the destination object. Must be the same data type as *src_lob*.

src_lob

An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator for the source object. Must be the same data type as *dest_lob*.

COMPARE function

The COMPARE function performs an exact byte-by-byte comparison of two large objects for a given length at given offsets.

The function returns:

- Zero if both large objects are exactly the same for the specified length for the specified offsets
- Non-zero if the objects are not the same
- Null if *amount*, *offset_1*, or *offset_2* are less than zero.

The large objects being compared must be the same data type.

Syntax

```
COMPARE(lob_1, lob_2 [ , amount [ , offset_1 [ , offset_2 ] ] ] )
```

Parameters

lob_1

An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the first large object to be compared. Must be the same data type as *lob_2*.

lob_2

An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the second large object to be compared. Must be the same data type as *lob_1*.

amount

An optional input argument of type INTEGER. If the data type of the large objects is BLOB, then the comparison is made for *amount* bytes. If the data type of the large objects is CLOB, then the comparison is made for *amount* characters. The default is the maximum size of a large object.

offset_1

An optional input argument of type INTEGER that specifies the position within the first large object to begin the comparison. The first byte (or character) is offset 1. The default is 1.

offset_2

An optional input argument of type INTEGER that specifies the position within the second large object to begin the comparison. The first byte (or character) is offset 1. The default is 1.

COPY procedures

The COPY procedures provide the capability to copy one large object to another.

The source and destination large objects must be the same data type.

Syntax

```
COPY_BLOB(dest_lob,src_lob,amount [ ,dest_offset [ ,src_offset ] ] )
```

```
COPY_CLOB(dest_lob,src_lob,amount [ ,dest_offset [ ,src_offset ] ] )
```

Parameters

dest_lob

An input or output argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to which *src_lob* is to be copied. Must be the same data type as *src_lob*.

src_lob

An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object from which *dest_lob* is to be copied. Must be the same data type as *dest_lob*.

amount

An input argument of type INTEGER that specifies the number of bytes or characters of *src_lob* to be copied.

dest_offset

An optional input argument of type INTEGER that specifies the position in the destination large object where writing of the source large object should begin. The first position is offset 1. The default is 1.

src_offset

An optional input argument of type INTEGER that specifies the position in the source large object where copying to the destination large object should begin. The first position is offset 1. The default is 1.

ERASE procedures

The ERASE procedures provide the capability to erase a portion of a large object.

To erase a large object means to replace the specified portion with zero-byte fillers for BLOBs or with spaces for CLOBs. The actual size of the large object is not altered.

Syntax

```
ERASE_BLOB(lob_loc,amount [ ,offset ] )
```

```
ERASE_CLOB(lob_loc,amount [ ,offset ] )
```

Parameters

lob_loc

An input or output argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to be erased.

amount

An input or output argument of type INTEGER that specifies the number of bytes or characters to be erased.

offset

An optional input argument of type INTEGER that specifies the position in the large object where erasing is to begin. The first byte or character is at position 1. The default is 1.

GETLENGTH function

The GETLENGTH function returns the length of a large object.

The function returns an INTEGER value that reflects the length of the large object in bytes (for a BLOB) or characters (for a CLOB).

Syntax

```
GETLENGTH(lob_loc)
```

Parameters

lob_loc

An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object whose length is to be obtained.

INSTR function

The INSTR function returns the location of the nth occurrence of a specified pattern within a large object.

The function returns an INTEGER value of the position within the large object where the pattern appears for the nth time, as specified by *nth*. This value starts from the position given by *offset*.

Syntax

```
INSTR( lob_loc, pattern [ ,offset [ ,nth ] ] )
```

Parameters

lob_loc

An input argument of type BLOB or CLOB that specifies the large object locator of the large object in which to search for the *pattern*.

pattern

An input argument of type BLOB(32767) or VARCHAR(32672) that specifies the pattern of bytes or characters to match against the large object.

pattern must be BLOB if *lob_loc* is a BLOB; and *pattern* must be VARCHAR if *lob_loc* is a CLOB.

offset

An optional input argument of type INTEGER that specifies the position within *lob_loc* to start searching for the *pattern*. The first byte or character is at position 1. The default value is 1.

nth

An optional argument of type INTEGER that specifies the number of times to search for the *pattern*, starting at the position given by *offset*. The default value is 1.

READ procedures

The READ procedures provide the capability to read a portion of a large object into a buffer.

Syntax

```
READ_BLOB(lob_loc,amount,offset,buffer)
```

```
READ_CLOB(lob_loc,amount,offset,buffer)
```

Parameters

lob_loc

An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to be read.

amount

An input or output argument of type INTEGER that specifies the number of bytes or characters to read.

offset

An input argument of type INTEGER that specifies the position to begin reading. The first byte or character is at position 1.

buffer

An output argument of type BLOB(32762) or VARCHAR(32672) that specifies the variable to receive the large object. If *lob_loc* is a BLOB, then *buffer* must be BLOB. If *lob_loc* is a CLOB, then *buffer* must be VARCHAR.

SUBSTR function

The SUBSTR function provides the capability to return a portion of a large object.

The function returns a BLOB(32767) (for a BLOB) or VARCHAR (for a CLOB) value for the returned portion of the large object read by the function.

Syntax

```
SUBSTR(lob_loc [ ,amount [ ,offset ] ] )
```

Parameters

lob_loc

An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to be read.

amount

An optional input argument of type INTEGER that specifies the number of bytes or characters to be returned. The default value is 32,767.

offset

An optional input argument of type INTEGER that specifies the position within the large object to begin returning data. The first byte or character is at position 1. The default value is 1.

TRIM procedures

The TRIM procedures provide the capability to truncate a large object to the specified length.

Syntax

```
TRIM_BLOB(lob_loc,newlen)
```

```
TRIM_CLOB(lob_loc,newlen)
```

Parameters

lob_loc

An input or output argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to be trimmed.

newlen

An input argument of type INTEGER that specifies the new number of bytes or characters to which the large object is to be trimmed.

WRITE procedures

The WRITE procedures provide the capability to write data into a large object.

Any existing data in the large object at the specified offset for the given length is overwritten by data given in the buffer.

Syntax

```
WRITE_BLOB(lob_loc,amount,offset,buffer)
```

```
WRITE_CLOB(lob_loc,amount,offset,buffer)
```

Parameters

lob_loc

An input or output argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to be written.

amount

An input argument of type INTEGER that specifies the number of bytes or characters in *buffer* to be written to the large object.

offset

An input argument of type INTEGER that specifies the offset in bytes or characters from the beginning of the large object for the write operation to begin. The start value of the large object is 1.

buffer

An input argument of type BLOB(32767) or VARCHAR(32672) that contains the data to be written to the large object. If *lob_loc* is a BLOB, then *buffer* must be BLOB. If *lob_loc* is a CLOB, then *buffer* must be VARCHAR.

DBMS_OUTPUT package

The DBMS_OUTPUT package provides a set of procedures for putting messages (lines of text) in a message buffer and getting messages from the message buffer. These procedures are useful during application debugging when you need to write messages to standard output.

The DBMS_OUTPUT package includes the following system-defined routines.

Table 29. System-defined routines available in the DBMS_OUTPUT package

Routine name	Description
DISABLE procedure on page 225	Disables the message buffer.
ENABLE procedure on page 225	Enables the message buffer.
GET_LINE procedure on page 226	Gets a line of text from the message buffer.
GET_LINES procedure on page 226	Gets one or more lines of text from the message buffer and places the text into a collection.
NEW_LINE procedure on page 227	Puts an end-of-line character sequence in the message buffer.
PUT procedure on page 227	Puts a string that includes no end-of-line character sequence in the message buffer.
PUT_LINE procedure on page 227	Puts a single line that includes an end-of-line character sequence in the message buffer.

Use the command line processor (CLP) command SET SERVEROUTPUT ON to redirect the output to standard output.

DISABLE and ENABLE procedures are not supported inside autonomous procedures.

An autonomous procedure is a procedure that, when called, executes inside a new transaction independent of the original transaction.

DISABLE procedure

The DISABLE procedure disables the message buffer.

After this procedure runs, any messages that are in the message buffer are discarded. Calls to the PUT, PUT_LINE, or NEW_LINE procedures are ignored, and no error is returned to the sender.

Syntax

```
DBMS_OUTPUT.DISABLE
```

ENABLE procedure

The ENABLE procedure enables the message buffer. During a single session, applications can put messages in the message buffer and get messages from the message buffer.

Syntax

```
DBMS_OUTPUT.ENABLE( buffer_size )
```

Procedure parameters

buffer_size

An input argument of type INTEGER that specifies the maximum length of the message buffer in bytes. If you specify a value of less than 2000 for *buffer_size*, the buffer size is set to 2000. If the value is NULL, then the default buffer size is 20000.

GET_LINE procedure

The GET_LINE procedure gets a line of text from the message buffer. The text must be terminated by an end-of-line character sequence.

Syntax

```
DBMS_OUTPUT.GET_LINE(line ,status)
```

Procedure parameters

line

An output argument of type VARCHAR(32672) that returns a line of text from the message buffer.

status

An output argument of type INTEGER that indicates whether a line was returned from the message buffer:

- `0` indicates that a line was returned
- `1` indicates that there was no line to return

GET_LINES procedure

The GET_LINES procedure gets one or more lines of text from the message buffer and stores the text in a collection. Each line of text must be terminated by an end-of-line character sequence.

Syntax

```
DBMS_OUTPUT.GET_LINES(lines ,numlines)
```

Procedure parameters

lines

An output argument of type DBMS_OUTPUT.CHARARR that returns the lines of text from the message buffer. The type DBMS_OUTPUT.CHARARR is internally defined as a VARCHAR(32672) ARRAY[2147483647] array.

numlines

An input and output argument of type INTEGER. When used as input, specifies the number of lines to retrieve from the message buffer. When used as output, indicates the actual number of lines that were retrieved from the message buffer. If the output value of *numlines* is less than the input value, then there are no more lines remaining in the message buffer.

NEW_LINE procedure

The NEW_LINE procedure puts an end-of-line character sequence in the message buffer.

Syntax

```
DBMS_OUTPUT.NEW_LINE
```

PUT procedure

The PUT procedure puts a string in the message buffer. No end-of-line character sequence is written at the end of the string.

Syntax

```
DBMS_OUTPUT.PUT(item)
```

Procedure parameters

item

An input argument of type VARCHAR(32672) that specifies the text to write to the message buffer.

PUT_LINE procedure

The PUT_LINE procedure puts a single line that includes an end-of-line character sequence in the message buffer.

Syntax

```
DBMS_OUTPUT.PUT_LINE(item )
```

Procedure parameters

item

An input argument of type VARCHAR(32672) that specifies the text to write to the message buffer.

DBMS_RANDOM package

The DBMS_RANDOM package provides a mechanism for generating random numbers. Use the INITIALIZE procedure to set the seed value, which is used by the random number generator to generate the numbers.

After enough repetitions, it is possible that some of the generated values will repeat. To reduce the possibility of repeating values, periodically change the seed value by using the SEED procedure.

The DBMS_RANDOM package includes the following system-defined routines and types.

Table 30. System-defined routines available in the DBMS_RANDOM package

Routine name	Description
INITIALIZE procedure on page 228	Initializes the package with the specified integer seed value. Optional.

Table 30. System-defined routines available in the DBMS_RANDOM package (continued)

Routine name	Description
SEED procedure on page 228	Resets the seed with the specified integer value.
RANDOM function on page 228	Uses the existing seed value to return a random integer.
TERMINATE procedure on page 229	Terminates the package by resetting the seed value to `0`. Optional.

INITIALIZE procedure

The INITIALIZE procedure initializes the system package with the specified integer seed value and is optional.

Syntax

```
DBMS_RANDOM_INITIALIZE ()
```

Example

This example:

```
execute procedure dbms_random_initialize (17809465);
```

Returns this output:

```
Routine executed.
```

SEED procedure

The SEED procedure resets the seed with the specified integer value.

Syntax

```
DBMS_RANDOM_SEED ()
```

Example

This example:

```
execute procedure dbms_random_seed (-45902345);
```

Returns this output:

```
Routine executed.
```

RANDOM function

The RANDOM function uses the seed value to return a random integer.

Syntax

```
DBMS_RANDOM_RANDOM ()
```

Example

This example:

```
insert into random_test VALUES (0, dbms_random_random());
```

Returns this output:

```
1 row(s) inserted.
```

TERMINATE procedure

The TERMINATE procedure terminates the use of the system package by resetting the seed value to `0` and is optional.

Syntax

```
DBMS_RANDOM_TERMINATE ()
```

Example

This example:

```
execute procedure dbms_random_terminate ();
```

Returns this output:

```
Routine executed.
```

UTL_FILE package

The UTL_FILE package provides a set of routines for reading from and writing to files on the database server file system.

The UTL_FILE system package includes the following system-defined routines and types.

Table 31. System-defined routines available in the UTL_FILE package

Routine name	Description
FCLOSE procedure on page 231	Closes a specified file.
FCLOSE_ALL procedure on page 231	Closes all open files.
FFLUSH procedure on page 231	Flushes unwritten data to a file.
FOPEN function on page 231	Opens a file.
GET_LINE procedure on page 232	Gets a line from a file.

Table 31. System-defined routines available in the UTL_FILE package (continued)

Routine name	Description
NEW_LINE procedure on page 232	Writes an end-of-line character sequence to a file.
PUT procedure on page 233	Writes a string to a file.

The following list describes all of the named conditions that an application can receive.

Table 32. Named conditions for an application

Condition Name	Description
access_denied	Access to the file is denied by the operating system.
charsetmismatch	A file was opened using FOPEN_NCHAR, but later I/O operations used non-CHAR functions such as PUTF or GET_LINE.
delete_failed	Unable to delete file.
file_open	File is already open.
internal_error	Unhandled internal error in the UTL_FILE system package.
invalid_filehandle	File handle does not exist.
invalid_filename	A file with the specified name does not exist in the path.
invalid_maxlinesize	The MAX_LINESIZE value for FOPEN is invalid. It must be 1 - 32672.
invalid_mode	The open_mode argument in FOPEN is invalid.
invalid_offset	The ABSOLUTE_OFFSET argument for FSEEK is invalid. It must be greater than 0 and less than the total number of bytes in the file.
invalid_operation	File could not be opened or operated on as requested.
invalid_path	The specified path does not exist or is not visible to the database.
read_error	Unable to read the file.
rename_failed	Unable to rename the file.
write_error	Unable to write to the file.

FCLOSE procedure

The FCLOSE procedure closes a specified file.

Syntax

```
UTL_FILE.FCLOSE(file)
```

Procedure parameters

file

An input or output argument of type UTL_FILE.FILE_TYPE that contains the file handle. When the file is closed, this value is set to 0.

FCLOSE_ALL procedure

The FCLOSE_ALL procedure closes all open files. The procedure runs successfully even if there are no open files to close.

Syntax

```
UTL_FILE.FCLOSE_ALL
```

FFLUSH procedure

The FFLUSH procedure forces unwritten data in the write buffer to be written to a file.

Syntax

```
UTL_FILE.FFLUSH(file)
```

Procedure parameters

file

An input argument of type UTL_FILE.FILE_TYPE that contains the file handle.

FOPEN function

The FOPEN function opens a file for I/O.

Syntax

```
UTL_FILE.FOPEN(location,filename,open_mode [ ,max_linesize ] )
```

Return value

This function returns a value of type UTL_FILE.FILE_TYPE that indicates the file handle of the opened file.

Function parameters

location

An input argument of type VARCHAR(128) that specifies the alias of the directory that contains the file.

filename

An input argument of type VARCHAR(255) that specifies the name of the file.

open_mode

An input argument of type VARCHAR(10) that specifies the mode in which the file is opened:

a

Append to file

r

Read from file

w

Write to file

max_linesize

An optional input argument of type INTEGER that specifies the maximum size of a line in characters. The default value is 1024 bytes. In read mode, an exception is thrown if an attempt is made to read a line that exceeds *max_linesize*. In write and append modes, an exception is thrown if an attempt is made to write a line that exceeds *max_linesize*. End-of-line characters do not count toward the line size.

GET_LINE procedure

The GET_LINE procedure gets a line of text from a specified file. The line of text does not include the end-of-line terminator. When there are no more lines to read, the procedure throws a NO_DATA_FOUND exception.

Syntax

```
UTL_FILE.GET_LINE(file, buffer)
```

Procedure parameters

file

An input argument of type UTL_FILE.FILE_TYPE that contains the file handle of the opened file.

buffer

An output argument of type VARCHAR(32672) that contains a line of text from the file.

NEW_LINE procedure

The NEW_LINE procedure writes an end-of-line character sequence to a specified file.

Syntax

```
UTL_FILE.NEW_LINE(file [ ,lines ] )
```

Procedure parameters

file

An input argument of type UTL_FILE.FILE_TYPE that contains the file handle.

lines

An optional input argument of type INTEGER that specifies the number of end-of-line character sequences to write to the file. The default is `1`.

PUT procedure

The PUT procedure writes a string to a specified file. No end-of-line character sequence is written at the end of the string.

Syntax

```
UTL_FILE.PUT(file, buffer)
```

Procedure parameters

file

An input argument of type UTL_FILE.FILE_TYPE that contains the file handle.

buffer

An input argument of type VARCHAR(32672) that specifies the text to write to the file.

Regex pattern matching

Regular expressions combine literal characters and metacharaters to define the search and replace criteria. You run the functions from the HCL OneDB™ Regex extension to find matches to strings, replace strings, and split strings into substrings.

The HCL OneDB™ Regex extension supports extended regular expressions, based on the POSIX 1003.2 standard, and basic regular expressions

You can specify case-sensitive or case-insensitive searching.

You can search single-byte character sets or UTF-8 character sets.

Henry Spencer's regular expression library

The Informix® Regex pattern matching utilizes Henry Spencer's regular expression library. Use of this library requires the following notice:

Copyright 1992, 1993, 1994, 1997 Henry Spencer. All rights reserved. This software is not subject to any license of the American Telephone and Telegraph Company or of the Regents of the University of California.

Permission is not granted to anyone to use this software for any purpose on any computer system, and to alter it and redistribute it, subject to the following restrictions:

1. The author is not responsible for the consequences of use of this software, no matter how awful, even if they arise from flaws in it.

2. The origin of this software must not be misrepresented, either by explicit claim or by omission. Since few users ever read sources, credits must appear in the documentation.

3. Altered versions must be plainly marked as such, and must not be misrepresented as being the original software. Since few users ever read sources, credits must appear in the documentation.

4. This notice may not be removed or altered.

Requirements and Restrictions

You must understand the following requirements and restrictions before you run regex pattern matching searches.

Regex support

You can use either basic or extended regular expressions, with case sensitivity or case insensitivity. Neither type of expression allows searching for a null character.

Extended regular expressions support more search and replace options than basic regular expressions.

Database server requirement

The Scheduler must be running in the database server. If the Scheduler is not running when you run regex functions, a message that the function is not found is returned.

You must have a default sbspace if you want to return a CLOB value when you replace strings.

Database requirements

The database where the regex functions are run must be logged and must not be an ANSI database. If you attempt to use a regex function in an unlogged or ANSI database, the message `DataBlade registration failed` is printed in the database server message log.

Data type support

To use regex pattern matching, you must provide the text data as a CHAR, LVARCHAR, NCHAR, NVARCHAR, VARCHAR, or CLOB data type.

If you want to replace text in a CLOB value with the regex_replace() function, you must have a default sbspace.

Locales and languages support

Regex queries can use single-byte character locales and UTF-8 based locales.

Important: If you use UTF-8 character encoding, including the Chinese GB18030-2000 code set, you must set the GL_USEGLU environment variable before you create the database.

Query restrictions

Regex functions do not inherently use any database indexes.

Metacharacters

A metacharacter is a character that has a special meaning during pattern processing. You use metacharacters in regular expressions to define the search criteria and any text manipulations.

Search string metacharacters are different from replacement string metacharacters.

Search string metacharacters

The following table lists metacharacters for extended regular expression searches.

If you use basic regular expressions, not all metacharacters are supported. The function of the backslash character is reversed. You must include a backslash character before all metacharacters.

Table 33. Search string metacharacters

Metacharacter	Action
^	Beginning of line
$	End of line
|	Or Not applicable to basic regular expressions.
[abc]	Match any character enclosed in the brackets
[^abc]	Match any character not enclosed in the brackets
[a-z]	Match the range of characters specified by the hyphen
[:*cclass*:]	Use the character list that is specified by *cclass*: • alnum = Uppercase and lowercase alphabetic characters and numbers: [A-Za-z0-9] • alpha = Uppercase and lowercase alphabetic characters: [A-Za-z] • blank = Whitespace and tab characters • cntrl = Control characters • digit = Numbers: [0-9] • graph = Visible characters (the alnum class plus the punct class) • lower = Lowercase alphabetic characters: [a-z] • print = Printable characters (the graph class plus whitespace) • punct = Punctuation marks: !"#$%&'()*+,-./:;<=>?@[\\]^_`{|}~

Table 33. Search string metacharacters (continued)

Metacharacter	Action
	• space = Whitespace characters: tab, newline, carriage-return, form-feed, and vertical-tab • upper = Uppercase alphabetic characters: [A-Z] • xdigit = Hexadecimal characters: [0-9a-fA-F] These classes are valid for single-byte character sets.
[=*cname*=]	Substitute the character name that is specified by *cname* with the corresponding character code. For a list of character names, see Regex character names on page 237.
.	Match any single character.
( )	Group the regular expression within the parentheses.
?	Match zero or one of the preceding expression. Not applicable to basic regular expressions.
*	Match zero, one, or many of the preceding expression.
+	Match one or many of the preceding expression. Not applicable to basic regular expressions.
\	Use the literal meaning of the metacharacter. For basic regular expressions, treat the next character as a metacharacter.

Replacement string metacharacters

The following table lists metacharacters for extended regular expression searches.

If you use basic regular expressions, not all metacharacters are supported. The function of the backslash character is reversed. You must include a backslash character before all metacharacters.

Table 34. Replacement string metacharacters

Metacharacter	Action
&	Reference the entire matched text for string substitution.

Table 34. Replacement string metacharacters (continued)

Metacharacter	Action
	For example, the statement `execute function regex_replace('abcdefg', '[af]', '.&.')` replaces `'a'` with `'.a.'` and `'f'` with `'.f.'` to return: `'.a.bcde.f.g'`.
\n	Reference the subgroup *n* within the matched text, where *n* is an integer 0-9. \0 and & have identical actions. \1 - \9 substitute the corresponding subgroup. For example, the statement `execute function regex_replace('abcdefg', '[af]', '.\0.')` replaces `'a'` with `'.a.'` and `'f'` with `'.f.'` to return: `'.a.bcde.f.g'`. For example, the statement `execute function regex_replace('abcdefg', '([af])([bg])', '.p1-\1.p2-\2.')` replaces `'ab'` with `'.p1-a.p2-b'` and `'fg'` with `'.p1-f.p2-g.'` to return: `'.p1-a.p2-b.cde.p1-f.p2-g.'`. Not applicable to basic regular expressions.
\	Use the literal meaning of the metacharacter, for example, `\&` escapes the Ampersand symbol and `\\` escapes the backslash. For basic regular expressions, treat the next character as a metacharacter.

Regex character names

You can search for character codes by specifying the character name in a regex search.

You must use the syntax [=*cname*=], where *cname* is a character name. The character code is determined in the following order:

1. If the character name exists in the current locale, the corresponding character code is used.
2. If the character name is one byte, the name is used as the code.
3. If the character name is listed in the following table, the corresponding character code is used.
4. Otherwise, the character name was not found, and an error is returned.

Table 35. Character names for regex searches

Name	Code
soh	01 = Start of heading
stx	02 = Start of text
etx	03 = End of text
eot	04 = End of transmission
enq	05 = Enquiry
ack	06 = Acknowledgment
bel	07 = Bell
alert	07 (BEL) = Bell
bs	08 = Backspace
backspace	08 = Backspace
ht	09 = Horizontal tab
tab	09 (HT) = Horizontal tab
lf	0A = Line feed
newline	0A (LF) = Line feed
vt	0B = Vertical tab
vertical-tab	0B (VT) = Vertical tab
ff	0C (FF) = Form feed
form-feed	0C (FF) = Form feed
cr	0D (CR) = Carriage return
carriage-return	0D (CR) = Carriage return
so	0E = Shift Out/X-On
si	0F = Shift In/X-Off
dle	10 = Data line escape
dc1	11 = Device control 1
dc2	12 = Device control 2
dc3	13 = Device control 3
dc4	14 = Device control 4
nak	15 = Negative acknowledgment

Table 35. Character names for regex searches (continued)

Name	Code
syn	16 = Synchronous idle
etb	17 = End of transmit block
can	18 = Cancel
em	19 = End of medium
sub	1A = Substitute
esc	1B = Escape
is4	1C (FS) = File separator
fs	1C = File separator
is3	1D (GS) = Group separator
gs	1D (GS) = Group separator
is2	1E (RS) = Record separator
rs	1E (RS) = Record separator
is1	1F (US) = Unit separator
us	1F (US) = Unit separator
space	' '
exclamation-mark	'!'
quotation-mark	""
number-sign	'#'
dollar-sign	'$'
percent-sign	'%'
ampersand	'&'
apostrophe	'\'
left-parenthesis	'('
right-parenthesis	')'
asterisk	'*'
plus-sign	'+'
comma	,
hyphen	'-'

Table 35. Character names for regex searches (continued)

Name	Code
hyphen-minus	-
period	'.'
full-stop	'.'
Slash	/
solidus	'/'
Zero	0
one	'1'
two	'2'
Three	3
four	4
Five	5
six	'6'
Seven	7
Eight	8
Nine	9
Colon	:
semicolon	';'
less-than-sign	'<'
equals-sign	'='
greater-than-sign	'>'
question-mark	?
commercial-at	@
left-square-bracket	'['
backslash	\
reverse-solidus	\
right-square-bracket	]
circumflex	^'
circumflex-accent	^'

Table 35. Character names for regex searches (continued)

Name	Code
underscore	_'
low-line	_'
grave-accent	`
left-brace	'{'
left-curly-bracket	'{'
vertical-line	'|'
right-brace	'}'
right-curly-bracket	'}'
Tilde	~
del	7F = Delete

Regex Routines

You can use regex routines to search for patterns, manipulate text strings, and configure tracing.

Example data

Some examples for the regex functions use the **tongue_twisters** table:

```
create table
      tongue_twisters  (     id
           int,     twister lvarchar  );
```

The **tongue_twisters** table contains the following data:

```
select id, twister
from   tongue_twisters
order by id;

id       246
twister  Sally sells sea shells by the sea shore. But if Sally sells sea
         shells by the sea shore then where are the sea shells Sally sells?

id       278
twister  Peter Piper picked a peck of pickled peppers. A peck of pickled
         peppers Peter Piper picked.  Peter Piper picked a peck of pickled
         peppers, Where's the peck of pickled peppers Peter Piper picked?

id       286
twister  If two witches would watch two watches, which witch would watch
         which watch?

id       301
twister  Fuzzy Wuzzy was a bear, Fuzzy Wuzzy had no hair, Fuzzy Wuzzy
```

```
         wasn't very fuzzy, was he?

id       306
twister  I slit a sheet, a sheet I slit, and on that slitted sheet I sit.

id       313
twister  Betty bought some bitter butter and it made her batter bitter, so
         Betty bought some better butter to make her bitter batter better.

id       335
twister  How much wood could a woodchuck chuck if a woodchuck could chuck
         wood? A woodchuck could chuck as much wood as a woodchuck would
         chuck if a woodchuck could chuck wood.

id       361
twister  She sells seashells on the seashore. The seashells she sells are
         seashore seashells.
```

regex_match function

The regex_match function returns indicates whether a source string matches the regular expression.

Syntax

```
regex_match(
     str     lvarchar,
     re      lvarchar,
     copts   integer DEFAULT 1)
returns boolean
```

```
regex_match(
     str     clob,
     re      lvarchar,
     copts_string   lvarchar)
returns boolean
```

Parameters

str

The string to search. Can be of type CHAR, NCHAR, VARCHAR, NVARCHAR, LVARCHAR or CLOB. A null value is treated as an empty string.

re

The regular expression. Can be of type CHAR, NCHAR, VARCHAR, NVARCHAR, or LVARCHAR. Cannot be null.

***copts* (Optional)**

The type of regex search:

- 0 = Basic regex
- 1 = Default. Extended POSIX regex

- 2 = Basic regex and ignore case
- 3 = Extended POSIX regex and ignore case

***copts_string* (Optional)**

The type of regex search:

- basic = Basic regex
- extended = Default. Extended POSIX regex
- basic,icase = Basic regex and ignore case
- extended,icase = Extended POSIX regex and ignore case
- basic,rtrim = Basic regex with rtrim
- extended,rtrim = Extended POSIX regex with rtrim
- basic,icase,rtrim = Basic regex and ignore case with rtrim
- extended,icase,rtrim = Extended POSIX regex and ignore case with rtrim

Description

Use the regex_match function to determine if the source string matches the regular expression.

Returns

t = The source string matches the regular expression.

f = The source string does not match the regular expression.

An exception = An error occurred.

Example

The following statement tests whether the word "module", "Module", or "DataBlade" occurs in the string:

```
execute function regex_match
('Regex module' , '[Mm]odule|DataBlade');
(expression) t
```

In the following example, the regular expression `'wo[ou]l?d'` matches the word "wood" and the word "would":

```
select id, twister
from     tongue_twisters
where    regex_match(twister, 'wo[ou]l?d');

id       286
twister  If two witches would watch two watches, which witch
         would watch which watch?
id       335
twister  How much wood could a woodchuck chuck if a
         woodchuck could chuck wood? A woodchuck
         could chuck as much wood as a woodchuck would
         chuck if a woodchuck could chuck wood.
```

regex_replace function

The regex_replace function replaces a string that matches a regular expression.

Syntax

```
regex_replace(
     str     lvarchar,
     re      lvarchar,
     rep     lvarchar,
     limit   integer DEFAULT 0,
     copts   integer DEFAULT 1)
returns lvarchar
```

```
regex_replace(
     str     clob,
     re      lvarchar,
     rep     lvarchar,
     limit   integer DEFAULT 0,
     copts_string   lvarchar)
returns clob
```

Parameters

str

The string to search. Can be of type CHAR, NCHAR, VARCHAR, NVARCHAR, LVARCHAR, or CLOB. A null value is treated as an empty string.

re

The regular expression. Can be of type CHAR, NCHAR, VARCHAR, NVARCHAR, or LVARCHAR. Cannot be null.

rep

The string to replace. Can be of type CHAR, NCHAR, VARCHAR, NVARCHAR, or LVARCHAR. See topic on Metacharacters on page 235 for metacharacter handling. Cannot be null.

***limit* (Optional)**

0 = Default. All matches are returned.

Positive integer = The maximum number of matches to return.

***copts* (Optional)**

The type of regex search:

- 0 = Basic regex
- 1 = Default. Extended POSIX regex
- 2 = Basic regex and ignore case
- 3 = Extended POSIX regex and ignore case

***copts_string* (Optional)**

The type of regex search:

- basic = Basic regex
- extended = Default. Extended POSIX regex
- basic,icase = Basic regex and ignore case
- extended,icase = Extended POSIX regex and ignore case
- basic,rtrim = Basic regex with rtrim
- extended,rtrim = Extended POSIX regex with rtrim
- basic,icase,rtrim = Basic regex and ignore case with rtrim
- extended,icase,rtrim = Extended POSIX regex and ignore case with rtrim

Description

Use the regex_replace function to replace text in a string. You can run the regex_replace function in an EXECUTE FUNCTION statement or in an SQL query, such as a SELECT statement.

Returns

A single value that is the input string with all substrings, up to the *limit* value, that match the input regular expression pattern replaced as specified by the replacement pattern.

An exception is returned if an error occurred.

Example

In this example, you want to have a web-based search engine that returns search matches in bold using the "and" HTML tags. Furthermore, you want to make the entire word bold in which the match was found. The regular expression in the example below looks for a word in which "she" or "She" occurs, then replaces the matched text with itself (&), enclosed by "and" HTML tags:

```
execute function regex_replace (
     'She sells seashells on the seashore. The seashells she sells are seashore
           seashells.',
     '( |^)[A-Za-z]*[Ss]he[a-z]*[.,$]',
     '<b>&</b>');
```

```
(expression) <b>She </b>sells<b> seashells </b>on the seashore. The<b>
seashells </b><b>she </b>sells are seashore<b> seashells.</b>
```

The result displayed on a web page looks like this:

```
She sells seashells on the seashore. The seashells she sells are seashore
seashells.
```

You can restrict the number of matches replaced by using the optional integer argument:

```
execute function regex_replace(
     'She sells seashells on the seashore. The seashells she sells are
     seashore seashells.',
     '( |^)[A-Za-z]*[Ss]he[a-z]*[ .,$]',
     '<b>&</b>',
       3);
```

The result displayed on a web page looks like this, with only three replacements:

```
She sells seashells on the seashore. The seashells she sells are seashore
seashells.
```

The following example runs the regex_replace function in a SELECT statement:

```
select id,
        regex_replace(twister, '( |^)[A-Za-z]*[Ss]he[a-z]*[ .,$]',
        '<b>&</b>')
from    tongue_twisters
where   regex_match(twister, '[Ss]he');
id            246
(expression)  Sally sells sea<b> shells </b>by the sea shore. But
              if Sally sells sea shells by the sea shore then
              where are the sea<b> shells </b>Sally sells?
id            306
(expression)  I slit a<b> sheet,</b> a<b> sheet </b>I slit, and
              on that slitted <b> sheet </b>I sit.
id            361
(expression)  <b>She </b>sells<b> seashells </b>on the seashore.
              The<b> seashells </b>she sells are seashore
              <b> seashells.</b>
```

The following statement reference four subgroups within the matched text:

```
execute function regex_replace (
        'swap me all around',
        '(.*) (.*) (.*) (.*)',
        '\4 \3 \2 \1'
);
(expression) around all me swap
1 row(s) retrieved.
execute function regex_replace ('swap me', '(.*) (.*)', '&: \2 \1');
(expression) swap me: me swap
1 row(s) retrieved.
```

regex_extract function

The regex_extract function returns a list of strings that match a regular expression from the source string.

Syntax

```
regex_extract(
     str      lvarchar,
     re       lvarchar,
     limit    integer DEFAULT 0,
     copts    integer DEFAULT 1)
returns lvarchar
```

```
regex_extract(
     str      clob,
     re       lvarchar,
     limit    integer DEFAULT 0,
     copts_string    lvarchar)
returns lvarchar
```

Parameters

str

The string to search. Can be of type CHAR, NCHAR, VARCHAR, NVARCHAR, LVARCHAR, or CLOB. A null value is treated as an empty string.

re

The regular expression. Can be of type CHAR, NCHAR, VARCHAR, NVARCHAR, or LVARCHAR. Cannot be null.

***limit* (Optional)**

0 = Default. All matches are returned.

Positive integer = The maximum number of matches to return.

***copts* (Optional)**

The type of regex search:

- 0 = Basic regex
- 1 = Default. Extended POSIX regex
- 2 = Basic regex and ignore case
- 3 = Extended POSIX regex and ignore case

***copts_string* (Optional)**

The type of regex search:

- basic = Basic regex
- extended = Default. Extended POSIX regex
- basic,icase = Basic regex and ignore case
- extended,icase = Extended POSIX regex and ignore case
- basic,rtrim = Basic regex with rtrim
- extended,rtrim = Extended POSIX regex with rtrim
- basic,icase,rtrim = Basic regex and ignore case with rtrim
- extended,icase,rtrim = Extended POSIX regex and ignore case with rtrim

Description

Use the regex_extract function iteratively return each substring that matches a regular expression. You can limit the number of substrings returned.

Returns

A set of text values that match the input regular expression pattern.

No rows found = A 0-length match, for example, a newline character.

An exception = An error occurred.

Example: Find a pattern within words

In this example, you want to find the patterns "would" and "wood":

```
How much wood could a
woodchuck chuck if a woodchuck could chuck wood?
A woodchuck could chuck as much wood as a woodchuck would chuck
if a woodchuck could chuck wood.
```

The following regular expression matches both the word "wood" and the word "would":

```
wo[ou]l?d
```

When you use this regular expression, the regex_extract function shows that the two words occur ten times in the string, but does not provide the entire word in which the pattern is found:

```
execute function
regex_extract(
     'How much wood could a woodchuck chuck if a woodchuck could chuck wood?
     A woodchuck could chuck as much wood as a woodchuck would chuck
     if a woodchuck could chuck wood.',
     'wo[ou]l?d'
);
(expression) wood
(expression) wood
(expression) wood
(expression) wood
(expression) wood
(expression) wood
(expression) wood
(expression) would
(expression) wood
(expression) wood
10 row(s) retrieved.
```

Example: Return the pattern plus the rest of the word

You can expand the regular expression to include the entire word. Start by specifying that more lowercase characters can follow the primary subexpression:

```
wo[ou]l?d[a-z]*
```

Next, add that the word ends with a space or a punctuation character. For completeness, you can specify more punctuation characters than the ones that occur in the text.

Note: If a hyphen appears as the first character in a character class, it means a literal hyphen, not a range of values.

```
wo[ou]l?d[a-z]*[- .?!:;]
```

When you run the regex_extract function with this expression, the function returns the whole words in which the pattern occurs:

```
execute function
regex_extract(
     'How much wood could a woodchuck chuck if a woodchuck could chuck wood?
```

```
     A woodchuck could chuck as much wood as a woodchuck would chuck
     if a woodchuck could chuck wood.',
     'wo[ou]l?d[a-z]*[- .?!:;]'
);
(expression) wood
(expression) woodchuck
(expression) woodchuck
(expression) wood?
(expression) woodchuck
(expression) wood
(expression) woodchuck
(expression) would
(expression) woodchuck
(expression) wood.
10 row(s) retrieved.
```

In the following example, you limit the results to the first two:

```
execute function
regex_extract(
     'How much wood could a woodchuck chuck if a woodchuck could chuck wood?
     A woodchuck could chuck as much wood as a woodchuck would chuck
     if a woodchuck could chuck wood.',
     'wo[ou]l?d[a-z]*[- .?!:;]',
     2
);
(expression) wood
(expression) woodchuck
2 row(s) retrieved.
```

Example: Return the pattern plus the beginning of the word

You can include the beginning of the word in the regular expression. The beginning of a word can be a space or the beginning of the line, (|^), followed by upper or lowercase letters:

```
( |^)[A-Za-z]*
```

In this example, you want to find all instances of the string "tter" in the following text:

```
Betty bought some bitter
butter and it made her batter bitter,
so Betty bought some better
butter to make her bitter batter better.
```

The following statement returns the pattern plus the beginnings of the words:

```
execute function
regex_extract(
     'Betty bought some bitter butter and it made her batter bitter,
     so Betty bought some better butter to make her bitter batter better.',
     '( |^)[A-Za-z]*tter'
);
(expression) bitter
(expression) butter
(expression) batter
(expression) bitter
(expression) better
(expression) butter
```

```
(expression) bitter
(expression) batter
(expression) better
9 row(s) retrieved.
```

Example: A match with 0 length

If the regular expression results in a 0-length match, the query returns the message "No rows found." For example, although the regex_match function returns t for a match on a begin-line character ("^"), the regex_extract function returns no rows because a search of "^" matches a string that has a length of 0:

```
execute function regex_match('Hello world', '^');
(expression) t
 1 row(s) retrieved.
execute function regex_extract('Hello world', '^');
 No rows found.
```

regex_split function

The regex_split function splits a string into substrings, using the match character as the delimiter.

Syntax

```
regex_split(
     str     lvarchar,
     re      lvarchar,
     limit   integer DEFAULT 0,
     copts   integer DEFAULT 1)
returns lvarchar
```

```
regex_split(
     str     clob,
     re      lvarchar,
     limit     integer DEFAULT 0,
     copts_string   lvarchar)
returns lvarchar
```

Parameters

str

The string to search. Can be of type CHAR, NCHAR, VARCHAR, NVARCHAR, LVARCHAR, or CLOB. A null value is treated as an empty string.

re

The regular expression. Can be of type CHAR, NCHAR, VARCHAR, NVARCHAR, or LVARCHAR. Cannot be null.

limit **(Optional)**

0 = Default. All matches are returned.

Positive integer = The maximum number of matches to return.

copts **(Optional)**

The type of regex search:

- 0 = Basic regex
- 1 = Default. Extended POSIX regex
- 2 = Basic regex and ignore case
- 3 = Extended POSIX regex and ignore case

***copts_string* (Optional)**

The type of regex search:

- basic = Basic regex
- extended = Default. Extended POSIX regex
- basic,icase = Basic regex and ignore case
- extended,icase = Extended POSIX regex and ignore case
- basic,rtrim = Basic regex with rtrim
- extended,rtrim = Extended POSIX regex with rtrim
- basic,icase,rtrim = Basic regex and ignore case with rtrim
- extended,icase,rtrim = Extended POSIX regex and ignore case with rtrim

Description

Use the regex_split function to split a string into substrings.

The regex_split function and the regex_extract function perform the complete opposite actions of each other.

Returns

A set of text values.

No rows found = The delimiter specified in the regular expression matches the entire source string.

An exception = An error occurred.

Example: Compare the regex_extract and regex_split functions

You are looking for the pattern "ick" and any characters that precede it:

```
( |^)[A-Za-z]*ick
```

The regex_extract function returns each substring that matches the regular expression:

```
execute function
     regex_extract(
     'Jack be nimble, Jack be quick, Jack jump over the candlestick.',
     '( |^)[A-Za-z]*ick'
);
(expression) quick
(expression) candlestick
2 row(s) retrieved.
```

The regex_split function splits the string into substrings, using the regular expression as the delimiter:

```
execute function
     regex_split(
     'Jack be nimble, Jack be quick, Jack jump over the candlestick.',
     '( |^)[A-Za-z]*ick');
(expression) Jack be nimble, Jack be
(expression) , Jack jump over the
(expression) .
3 row(s) retrieved.
```

Example: Split a string into separate words

The following example splits the string up into its separate words, using a space as the delimiter:

```
execute function
     regex_split(
     'Jack be nimble, Jack be quick, Jack jump over the candlestick.',
     ' ');
(expression) Jack
(expression) be
(expression) nimble,
(expression) Jack
(expression) be
(expression) quick,
(expression) Jack
(expression) jump
(expression) over
(expression) the
(expression) candlestick.
11 row(s) retrieved.
```

The following example limits the number of substrings into which the source string is split to 5:

```
execute function
     regex_split(
     'Jack be nimble, Jack be quick, Jack jump over the candlestick.',
     ' ',
     5);
(expression) Jack
(expression) be
(expression) nimble,
(expression) Jack
(expression) be quick, Jack jump over the candlestick.
5 row(s) retrieved.
```

Example: The regular expression matches the entire source string

In the following example, the delimiter specified in the regular expression matches the entire source string and returns "No rows found.":

```
execute function regex_split('Hello world', 'Hello world');
No rows found.
```

regex_set_trace procedure

The regex_set_trace() procedure enables the output of debug messages.

Syntax

```
regex_set_trace(
     filename     lvarchar,
     level        integer)
```

Parameters

filename

The file name for the trace entries. The file must be accessible to the database server.

level

The trace level: 10, 20, 30, or 100. Higher values return more details.

Description

The regex_set_trace() procedure enables the output of debug messages.

Example

The following statement enables tracing:

```
Execute procedure regex_set_trace('/tmp/rx.log', 20);
```

regex_release function

The regex_release function provides the internal release version number of the regex extension.

Syntax

```
regex_release()
returns lvarchar
```

Parameters

None

Description

Run the regex_release function if HCL Software Support asks for the internal release number.

Returns

The internal release version number.

An exception = An error occurred.

Example

```
execute function regex_release();
(expression) ifxregex release 1.00 (Build 184)
             Compiled on Sat Jul 2 08:25:49 CDT 2016 with:
```

```
                IBM Informix Dynamic Server Version 12.10.FC8
                glslib-6.00.UC8
```

Index