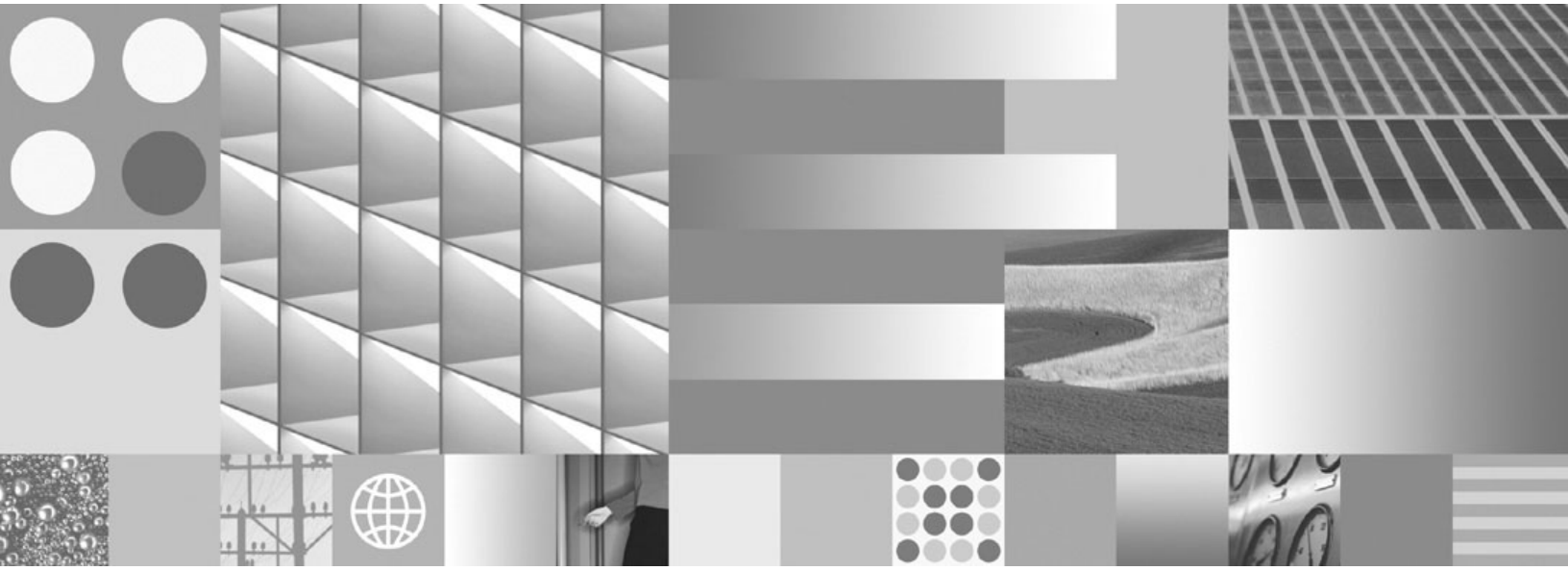
IBM Informix

Version 11.50

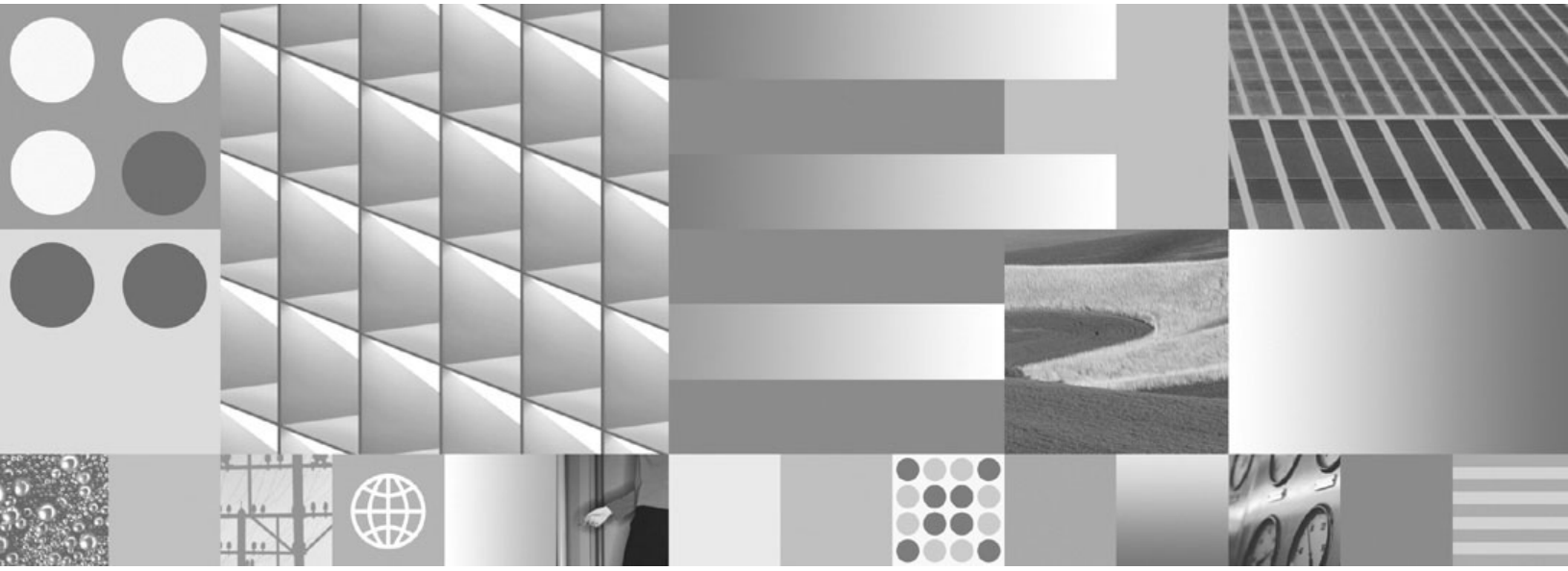


IBM Informix Database Extensions User's Guide

SC23-9427-02

IBM Informix

Version 11.50



IBM Informix Database Extensions User's Guide

SC23-9427-02

Note:
Before using this information and the product it supports, read the information in "Notices" on page B-1.

This edition replaces SC23-9427-01.

Contents

Part 3. Binary Data Types

Part 4. Basic Text Search

Introduction

In This Introduction

This introduction introduces the *IBM Informix Database Extensions User's Guide*. Read this chapter for an overview of the information provided in this publication and for an understanding of the conventions used throughout.

About This Publication

This publication explains how to use the following IBM® Informix® DataBlade® modules that come with IBM Informix Dynamic Server:

- Large Object Locator, a foundation DataBlade module for large objects management that can be used by other modules that create or store large-object data.
- MQ DataBlade module, which allows IBM Informix database applications to communicate with other MQSeries® applications with MQ messaging.
- Binary DataBlade Module that includes binary data types that allow you to store binary-encoded strings, which can be indexed for quick retrieval.
- Basic Text Search DataBlade module, which allows you to search words and phrases stored in a column of a table.
- Node DataBlade Module for the hierarchical data type, which along with its supporting functions, gives you the ability to represent hierarchical data within the relational database.
- Web Feature Service DataBlade module, which lets you add an Open Geospatial Consortium (OGC) web feature service as a presentation layer for the Spatial and Geodetic DataBlade modules.

Types of Users

This publication is for application developers and database administrators who want to use the built-in extensions provided in Informix Dynamic Server for storing, querying, and manipulating data.

What's New in Database Extensions for Dynamic Server, 11.50

Version 11.50 includes new features for the Basic Text Search DataBlade module. For a comprehensive list of new features for this release, see the *IBM Informix Dynamic Server Getting Started Guide*. The following changes and enhancements are relevant to this publication.

Table 1. What's New in the IBM Informix Database Extensions User's Guide for Version 11.50.xC4

Overview	Reference
Control the Results of a Fuzzy Search with the Basic Text Search DataBlade Module You can now specify the degree of similarity of search results in fuzzy searches when using the Basic Text Search DataBlade module. Specify a number between 0 and 1, where a higher value results in a higher degree of similarity. To limit results, specify a higher number. To maximize results, specify a lower number. The default degree of similarity is 0.5.	"Fuzzy Searches" on page 14-4
Map Characters for Indexing with the Basic Text Search DataBlade Module You can now map characters in your data to other characters during indexing with the Basic Text Search DataBlade module. For example, you can specify that letters with diacritical marks are indexed as the same letters without marks. You can also standardize inconsistent prefixes or delete character strings from indexed text. To use character maps, include the **canonical_maps** parameter when you create your **bts** index.	"Canonical Mapping" on page 14-9
Default Boolean Operator in Basic Text Search Queries You can now change the default Boolean operator between search terms in Basic Text Search queries from OR to AND by using the **query_default_operator** parameter when you create a **bts** index. The default operator is represented by a blank space between terms. Many popular end-user search engines use AND as the default operator between search terms, where end-users expect the search results to contain all their search terms.	"Boolean Operators" on page 14-5
Storage for Temporary Basic Text Search Files You can now specify that temporary files used by the Basic Text Search DataBlade module are stored in a separate sbspace from the one used to store the **bts** index. Separating temporary files from the **bts** index might improve query performance.	"Creating a Space for Temporary Data" on page 13-5
Track Basic Text Search Query Trends You can now track what queries are run against your **bts** index by including the **query_log** parameter when you create a **bts** index. You can use query trends information to provide hints to end-users on popular queries or work on optimizing the most popular queries.	"Tracking Queries on bts Indexes" on page 13-7
Fragment **bts** Indexes by Expressions You can now fragment **bts** indexes by expressions into multiple sbspaces instead of a single sbspace.	"Creating the Index by Specifying the bts Access Method" on page 13-5

Table 2. What's New in the IBM Informix Database Extensions User's Guide for Version 11.50.xC3

Overview	Reference
Basic Text Search DataBlade Module Supports High-Availability Clusters You can now use the Basic Text Search DataBlade module to perform searches on high-availability cluster servers by creating indexes in sbspaces. Previously, the Basic Text Search DataBlade module only supported the creation of indexes in extspaces, and thus could not participate in any queries on high-availability secondary servers and in backup and restore operations.	"Creating a Space for the bts Index" on page 13-4
Querying XML Attributes with the Basic Text DataBlade Module The Basic Text Search DataBlade Module now supports searches on XML attributes in a document repository. The new **all_xmlattrs** parameter enables searches on all attributes that are contained in the XML tags or paths in a column that contains an XML document.	"The all_xmlattrs Index Parameter" on page 15-6

Table 3. What's New in the IBM Informix Database Extensions User's Guide for Version 11.50.xC1

Overview	Reference
Support added for a user-defined stopword list	"Basic Text Search Stopwords" on page 14-7
Support added for XML-structured documents	Chapter 15, "Basic Text Search XML Index Parameters," on page 15-1

Documentation Conventions

Special conventions are used in the product documentation for IBM Informix Dynamic Server.

Technical Changes

Technical changes to the text are indicated by special characters depending on the format of the documentation.

HTML documentation

New or changed information is surrounded by blue >> and << characters.

PDF documentation

A plus sign (+) is shown to the left of the current changes. A vertical bar (|) is shown to the left of changes made in earlier shipments.

Feature, Product, and Platform Markup

Feature, product, and platform markup identifies paragraphs that contain feature-specific, product-specific, or platform-specific information.

Some examples of this markup follow:

Dynamic Server

Identifies information that is specific to IBM Informix Dynamic Server

End of Dynamic Server

Windows Only

Identifies information that is specific to the Windows operating system

End of Windows Only

This markup can apply to one or more paragraphs within a section. When an entire section applies to a particular product or platform, this is noted as part of the heading text, for example:

Table Sorting (Windows)

Example Code Conventions

Examples of SQL code occur throughout this publication. Except as noted, the code is not specific to any single IBM Informix application development tool.

If only SQL statements are listed in the example, they are not delimited by semicolons. For instance, you might see the code in the following example:

```
CONNECT TO stores_demo
...

DELETE FROM customer
   WHERE customer_num = 121
...

COMMIT WORK
DISCONNECT CURRENT
```

To use this SQL code for a specific product, you must apply the syntax rules for that product. For example, if you are using an SQL API, you must use EXEC SQL at the start of each statement and a semicolon (or other appropriate delimiter) at the end of the statement. If you are using DB–Access, you must delimit multiple statements with semicolons.

Tip: Ellipsis points in a code example indicate that more code would be added in a full application, but it is not necessary to show it to describe the concept being discussed.

For detailed directions on using SQL statements for a particular application development tool or SQL API, see the documentation for your product.

Additional Documentation

Documentation about IBM Informix products is available in various formats.

You can view, search, and print all of the product documentation from the IBM Informix Dynamic Server information center on the Web at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp.

For additional documentation about IBM Informix Dynamic Server and related products, including release notes, machine notes, and documentation notes, go to the online product library page at http://www.ibm.com/software/data/informix/pubs/library/. Alternatively, you can access or install the product documentation from the Quick Start CD that is shipped with the product.

Compliance with Industry Standards

IBM Informix products are compliant with various standards.

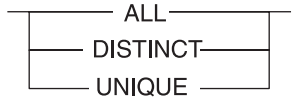
The American National Standards Institute (ANSI) and the International Organization of Standardization (ISO) have jointly established a set of industry standards for the Structured Query Language (SQL). IBM Informix SQL-based products are fully compliant with SQL-92 Entry Level (published as ANSI X3.135-1992), which is identical to ISO 9075:1992. In addition, many features of IBM Informix database servers comply with the SQL-92 Intermediate and Full Level and X/Open SQL Common Applications Environment (CAE) standards.

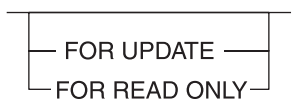
Syntax Diagrams

Syntax diagrams use special components to describe the syntax for statements and commands.

Table 4. Syntax Diagram Components

Component represented in PDF	Component represented in HTML	Meaning
	`>>----------`	Statement begins.
	`---------->`	Statement continues on next line.
	`>----------`	Statement continues from previous line.
	`----------><`	Statement ends.
	`--------SELECT----------`	Required item.
	`--+-----------------+---` `  '------LOCAL------'`	Optional item.

	`---+-----ALL-------+---`
	`   +--DISTINCT-----+`
	`   '---UNIQUE------'`
		Required item with choice. One and only one item must be present.

	`---+------------------+---`
	`    +--FOR UPDATE-----+`
	`    '--FOR READ ONLY--'`
		Optional items with choice are shown below the main line, one of which you might specify.

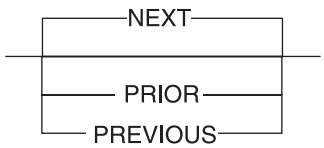
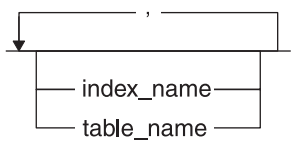
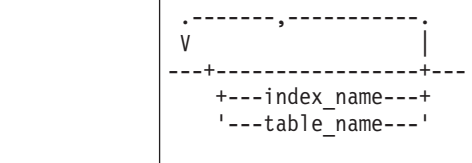
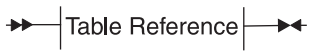
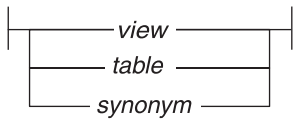
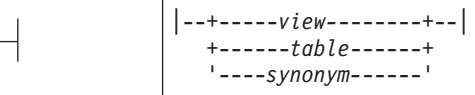
	`    .---NEXT---------.`
	`----+----------------+---`
	`    +---PRIOR--------+`
	`    '---PREVIOUS-----'`
		The values below the main line are optional, one of which you might specify. If you do not specify an item, the value above the line will be used as the default.

Table 4. Syntax Diagram Components (continued)

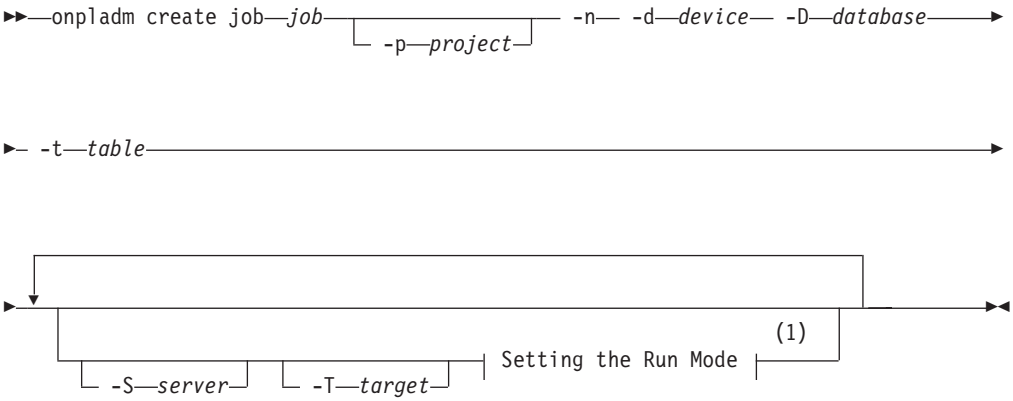
Component represented in PDF	Component represented in HTML	Meaning
		Optional items. Several items are allowed; a comma must precede each repetition.
		Reference to a syntax segment.
Table Reference 	Table Reference 	Syntax segment.

How to Read a Command-Line Syntax Diagram

Command-line syntax diagrams use similar elements to those of other syntax diagrams.

Some of the elements are listed in the table in Syntax Diagrams.

Creating a No-Conversion Job

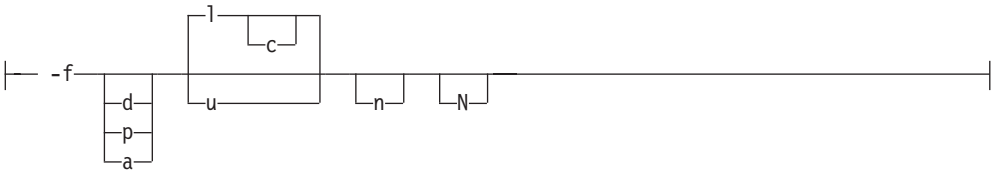


Notes:

1 See page Z-1

This diagram has a segment named "Setting the Run Mode," which according to the diagram footnote is on page Z-1. If this was an actual cross-reference, you would find this segment in on the first page of Appendix Z. Instead, this segment is shown in the following segment diagram. Notice that the diagram uses segment start and end components.

Setting the Run Mode:



To see how to construct a command correctly, start at the top left of the main diagram. Follow the diagram to the right, including the elements that you want. The elements in this diagram are case sensitive because they illustrate utility syntax. Other types of syntax, such as SQL, are not case sensitive.

The Creating a No-Conversion Job diagram illustrates the following steps:

1. Type **onpladm create job** and then the name of the job.
2. Optionally, type **-p** and then the name of the project.
3. Type the following required elements:
   - **-n**
   - **-d** and the name of the device
   - **-D** and the name of the database
   - **-t** and the name of the table
4. Optionally, you can choose one or more of the following elements and repeat them an arbitrary number of times:
   - **-S** and the server name
   - **-T** and the target server name
   - The run mode. To set the run mode, follow the Setting the Run Mode segment diagram to type **-f**, optionally type **d**, **p**, or **a**, and then optionally type **l** or **u**.
5. Follow the diagram to the terminator.

Keywords and Punctuation

Keywords are words reserved for statements and all commands except system-level commands.

When a keyword appears in a syntax diagram, it is shown in uppercase letters. When you use a keyword in a command, you can write it in uppercase or lowercase letters, but you must spell the keyword exactly as it appears in the syntax diagram.

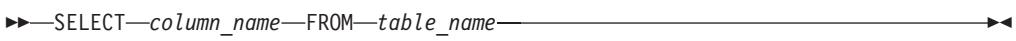
You must also use any punctuation in your statements and commands exactly as shown in the syntax diagrams.

Identifiers and Names

Variables serve as placeholders for identifiers and names in the syntax diagrams and examples.

You can replace a variable with an arbitrary name, identifier, or literal, depending on the context. Variables are also used to represent complex syntax elements that are expanded in additional syntax diagrams. When a variable appears in a syntax diagram, an example, or text, it is shown in *lowercase italic*.

The following syntax diagram uses variables to illustrate the general form of a simple SELECT statement.



When you write a SELECT statement of this form, you replace the variables *column_name* and *table_name* with the name of a specific column and table.

How to Provide Documentation Feedback

You are encouraged to send your comments about IBM Informix user documentation.

Use one of the following methods:

- Send e-mail to docinf@us.ibm.com.
- Go to the information center at http://publib.boulder.ibm.com/infocenter/idshelp/v115/index.jsp and open the topic that you want to comment on. Click the feedback link at the bottom of the page, fill out the form, and submit your feedback.

Feedback from both methods is monitored by those who maintain the user documentation. The feedback methods are reserved for reporting errors and omissions in our documentation. For immediate help with a technical problem, contact IBM Technical Support. For instructions, see the IBM Informix Technical Support Web site at http://www.ibm.com/planetwide/.

We appreciate your suggestions.

Part 1. Large Objects Management

Chapter 1. About Large Object Locator

In This Chapter

This chapter provides an overview of the IBM Informix Large Object Locator DataBlade Module.

Large Object Locator 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.

Informix Dynamic Server 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 *IBM Informix Guide to SQL: Syntax* and *IBM Informix 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 pathname 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.

Using Large Object Locator

Large Object Locator is implemented through two data types and a set of functions, described next.

Large Object Locator Data Types

Large Object Locator defines two data types, 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 "lld_locator" on page 2-1 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 "lld_lob" on page 2-2 for a complete description of this data type.

Large Object Locator Functions

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** for creating, opening, closing, deleting, and reading from and writing to large objects.
- **Client functions** for 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** for raising errors and converting errors to their SQL state equivalents.
- **Smart large object functions** for 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 3-2 for a list of the Large Object Locator functions that you can execute directly in SQL statements.

Chapter 3, "Large Object Locator Functions," on page 3-1, describes all the Large Object Locator functions and the three interfaces in detail.

Limitations

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.

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.

Concurrent Access

For the same reason, 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.

Installation and Registration

Large Object Locator is distributed with Informix Dynamic Server. To use the Large Object Locator functions, you must use BladeManager to register the functions and data types with each database for which you want Large Object Locator functionality. See the *IBM Informix DataBlade Module Installation and Registration Guide* for more information. This guide also contains some information about installing DataBlade modules.

Chapter 2. Large Object Locator Data Types

In This Chapter

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

lld_locator

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 is 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 is 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 2-2.

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

Table 2-1 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 2-1. 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 case insensitive. 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 *IBM Informix ESQL/C Programmer's Manual* for information on row types. See the *IBM Informix DataBlade API Programmer's Guide* for information on 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 Chapter 3, "Large Object Locator Functions," on page 3-1:

- "lld_copy()" on page 3-5
- "lld_create()" on page 3-7
- "lld_delete()" on page 3-9
- "lld_open()" on page 3-10
- "lld_from_client()" on page 3-20
- "lld_to_client()" on page 3-24

lld_lob

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 is a pointer to the location of the smart large object.

type is 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 "Using the lld_lob Type" on page 4-1 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 Chapter 3, "Large Object Locator Functions," on page 3-1:

- "LOCopy" on page 3-29
- "LOToFile" on page 3-30
- "LLD_LobType" on page 3-31

Note that **LOCopy** and **LOToFile** 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.

Chapter 3. Large Object Locator Functions

In This Chapter

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 IBM Informix DataBlade API. See the *IBM Informix DataBlade API Programmer's Guide*for more information.

See "Using the API" on page 4-7 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 *IBM Informix ESQL/C Programmer's Manual* for more information on 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 IBM Informix SQL interface:

- *IBM Informix Guide to SQL: Reference*
- *IBM Informix Guide to SQL: Syntax*
- *IBM Informix 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).

   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()** to return the current position within the large object.
   - **lld_seek()** to change the current position within the object.
   - **lld_read()** to read from large object.
   - **lld_write()** to write to the large object.
   - **lld_close()** to close 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.

lld_close()

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

error is 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

"lld_open()" on page 3-10

lld_copy()

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 is 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 is a pointer to the lld_locator row, identifying the source object.

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

error is 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

"lld_from_client()" on page 3-20

"lld_to_client()" on page 3-24

lld_create()

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 is 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 is a pointer to an lld_locator row, identifying the object to create.

error is 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 "Transaction Rollback" on page 1-3 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

"lld_delete()" on page 3-9

"lld_open()" on page 3-10

lld_delete()

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 is 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 is a pointer to an lld_locator row, identifying the object to delete.

error is 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.

lld_open()

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 is 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.

lob is a pointer to an lld_locator row, identifying the object to open.

flags is 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 is 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 "Limitations" on page 1-3 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

"lld_close()" on page 3-4

"lld_create()" on page 3-7

"lld_read()" on page 3-12

"lld_seek()" on page 3-13

"lld_tell()" on page 3-15

"lld_write()" on page 3-16

lld_read()

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

buffer is 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 is the number of bytes to read.

error is 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

"lld_open()" on page 3-10

"lld_seek()" on page 3-13

"lld_tell()" on page 3-15

lld_seek()

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 is 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. The ESQL/C version of this function is based on the assumption that you are already connected to a server.

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

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

error is 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

"lld_open()" on page 3-10

"lld_read()" on page 3-12

"lld_tell()" on page 3-15

"lld_write()" on page 3-16

lld_tell()

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

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

error is 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

"lld_open()" on page 3-10

"lld_read()" on page 3-12

"lld_seek()" on page 3-13

"lld_write()" on page 3-16

lld_write()

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

buffer is 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 is the number of bytes to write.

error is 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

"lld_open()" on page 3-10

"lld_seek()" on page 3-13

"lld_tell()" on page 3-15

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 filenames in the database might find at a later date that the files are inaccessible.

lld_create_client()

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 is 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.

path is a pointer to the pathname of the client file.

error is 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 "Transaction Rollback" on page 1-3 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

"lld_delete_client()" on page 3-19

lld_delete_client()

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 is 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.

path is a pointer to the pathname of the client file.

error is 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

"lld_create_client()" on page 3-18

lld_from_client()

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 is 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 is a pointer to the source pathname.

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

error is 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

"lld_create_client()" on page 3-18

"lld_open_client()" on page 3-22

lld_open_client()

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 is 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.

path is a pointer to the path of the client file to open.

flags is 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 is 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 "Limitations" on page 1-3 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

"lld_close()" on page 3-4

"lld_read()" on page 3-12

"lld_seek()" on page 3-13

"lld_tell()" on page 3-15

"lld_write()" on page 3-16

"lld_create_client()" on page 3-18

lld_to_client()

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 is 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 is a pointer to the lld_locator row that identifies the source large object.

dest is a pointer to the destination pathname. If the destination file does not exist, it is created.

error is 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

"lld_open_client()" on page 3-22

Error Utility Functions

The two functions described in this section allow you to:

- raise error exceptions.
- convert error codes to their SQL state equivalent.

lld_error_raise()

This function generates an exception for the specified error.

Syntax

API:

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

error is 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`.

lld_sqlstate()

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 is 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.

LOCopy

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 is a pointer to the smart large object to copy.

table_name is a table name. This parameter is optional.

column_name is 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 *IBM Informix 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 *IBM Informix Guide to SQL: Syntax*

LOToFile

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 is a pointer to the smart large object.

pathname is a directory path and name of the file to create.

file_dest is 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 *IBM Informix Guide to SQL: Syntax* for complete information about this function.

Return codes

This function returns the value of the new filename.

Context

LOToFile in the *IBM Informix Guide to SQL: Syntax*

LLD_LobType

Returns the type of data in an lld_lob column.

Syntax

SQL:

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

lob is 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.

Chapter 4. Large Object Locator Example Code

In This Chapter

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.

Using the SQL Interface

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

Using 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.

Using Implicit lld_lob Casts

This section shows how to insert binary and character data into an lld_lob type column of a table. The example in Figure 4-1 makes use of implicit casts from BLOB and CLOB types to the lld_lob type.

```
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
     460736c6f000010029a2a6c930d0000000000006c000af0cdd900000016000000010af0c9d
     e
```

Figure 4-1. Implicit lld_lob Casts

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.

Using Explicit lld_lob Casts

The example in Figure 4-2 on page 4-3 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 4-1 on page 4-2.

```
--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.
```

Figure 4-2. Explicit lld_lob Casts

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.

Using the LLD_LobType Function

The example in this section, Figure 4-3, 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 4-1 on page 4-2. That example created the table and inserted a BLOB type large object for key 1 and a CLOB type large object for key 2.

```
-- 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.
```

Figure 4-3. Using LLD_LobType Function

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.

Using 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 is 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 is 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 for an existing server file into a table.
- create a smart large object.
- copy a client file to a large object.
- copy a large object to another large object.
- copy a large object to a client file.
- create and delete a server file.

Inserting an lld_locator Row into a Table

The example in Figure 4-4 creates a table with an lld_locator row and shows how to insert a large object into the row.

```
--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')");
```

Figure 4-4. Inserting an lld_locator Row Into a Table

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**.

Creating a Smart Large Object

The example in Figure 4-5 creates a smart large object containing CLOB type data. The **lld_create** function in Figure 4-5 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**.

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

Figure 4-5. Using lld_create

Copying a Client File to a Large Object

The example in Figure 4-6 uses the **lobs** table created in Figure 4-5.

In Figure 4-6, the **lld_fromclient** function in the first SELECT statement, copies the client file, **quote2.txt**, to an lld_locator row in the **lobs** table.

```
-- 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
          00000005c4e6000607fdc00000000000000000000000000',NULL)


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


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

Figure 4-6. Copying a Client File to a Large Object

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.

Copying a Large Object to a Large Object

The example in Figure 4-7 uses the **lobs** table created in Figure 4-4 on page 4-4.

The **lld_copy** function in Figure 4-7 copies large object data from one lld_locator type row to another.

```
-- 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
          00000005c4e6000607fdc00000000000000000000000000',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.
```

Figure 4-7. Copying a Large Object to a Large Object

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

Copying Large Object Data to a Client File

The example in Figure 4-8 uses the **lobs** table created in Figure 4-4 on page 4-4. The **lld_toclient** function in "Copying Large Object Data to a Client File" on page 4-6 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.

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

(expression)

          t
```

Figure 4-8. Copying a Large Object to a Client File

Creating and Deleting a Server File

The example in Figure 4-9 on page 4-7 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.

```
-- 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;
```

Figure 4-9. Creating and Deleting a Server File

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.

Using the API

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.

Creating the lld_copy_subset Function

Figure 4-10 on page 4-8 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 4-10. The lld_copy_subset Function (Part 1 of 2)

```
    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;
}
```

Figure 4-10. The lld_copy_subset Function (Part 2 of 2)

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**, to open the source object
- **lld_seek**, to seek to the specified byte offset in the object
- **lld_read**, to read the specified number of bytes from the object
- **lld_close**, to close 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 *IBM Informix DataBlade API Programmer's Guide* for more information.

Using the lld_copy_subset Routine

The example in this section, Figure 4-11, shows how to use the **lld_copy_subset** user-defined routine defined in the previous section.

```
-- 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;
```

Figure 4-11. Using the lld_copy_subset Routine

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.

Chapter 5. Large Object Locator Error Handling

In This Chapter

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.

Handling 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.

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 IBM Informix 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.
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.

Part 2. MQ Messaging

Chapter 6. About the MQ DataBlade Module

In This Chapter

This chapter describes how to prepare, verify, and use the IBM Informix MQ DataBlade module.

About IBM WebSphere MQ and MQ DataBlade Module

IBM WebSphere® MQ (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 IBM Informix MQ DataBlade (MQ DataBlade) module provides the functionality to exchange messages between Informix Dynamic Server (Dynamic Server) databases and WMQ message queues.

Using MQ DataBlade Tables and Functions

The MQ DataBlade module uses either functions or tables to communicate between a Dynamic Server application and an MQ queue. For more information on using MQ DataBlade functions, see Chapter 8, “MQ DataBlade Functions.” For more information on MQ DataBlade tables, see Chapter 7, “MQ DataBlade Tables.”

Limitations

MQ DataBlade module has the following limitations:

- Each Dynamic Server instance can connect to only one WMQ queue manager.
- Dynamic Server and WMQ products must be installed on the same machine.
- Non-logged databases are not supported.
- Registration to an ANSI database is not supported.

Software Requirements

To use MQ DataBlade module, the following software must be installed:

- IBM WebSphere MQ, Version 5.3
- IBM Informix Dynamic Server, Version 10.00.xC3 and above

Note: IBM Informix MQ DataBlade module, Version 2.0 is distributed with Dynamic Server 10.00.xC3 and above.

Note: WMQ platform requirements are independent of Dynamic Server platform requirements. For more information on respective platform requirements, see the WMQ documentation and Dynamic Server machine notes.

Preparing to use MQ DataBlade Module

The MQ DataBlade module is included with Dynamic Server. When you install Dynamic Server, the MQ DataBlade module is installed automatically.

Before you can use the MQ DataBlade module, you must complete the following tasks, each of which is explained below:

1. Install WMQ.
2. Configure that WMQ queues.
3. Configure Dynamic Server.
4. Register the MQ DataBlade module.

Installing WMQ

You must install IBM WebSphere MQ, Version 5.3 before using the MQ DataBlade module. Information on how to install WMQ is included in the WMQ product documentation.

Configuring the WMQ Queues

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.

Configuring Dynamic Server

Dynamic Server database requires the following configuration steps, each of which is explained below:

1. Add the user **informix** to the **mqm** group.
2. Create a new virtual processor class called **mq**.
3. Review the "**informix**".**mqi*** tables.

Add the User informix to the mqm Group

Only members of the **mqm** group are authorized to access to WMQ queues. User **informix** must be made a member of the **mqm** group. For information on how to add user **informix** to the **mqm** group, see the platform-specific documentation for WMQ.

After adding the user **informix** to the **mqm** group and before continuing with the next step, you must shut down and restart Dynamic Server.

Creating the mq Virtual Processor Class

The VPCLASS parameter of the Dynamic Server ONCONFIG file allows you to create a class of virtual processors. You must create an **mq** VP class with the **noyield** option. Create the VP class in one of the following two ways:

- Run

  ```
  onmode -p +1 mq
  ```

- Manually change the ONCONFIG file, which is located in the **$INFORMIXDIR/etc** directory. Add the following parameter to your ONCONFIG file:

  ```
  VPCLASS mq,noyield,num=1
  ```

If you manually change the ONCONFIG file, you must shut down and restart Dynamic Server for these changes to be effective.

Note: If the VP class is not created, the following error will be returned when you attempt to use the MQ DataBlade module:

```
9799: User define routine (mqread) VP context switch failed.
```

Reviewing the "informix".mqi* Tables

During registration of MQ DataBlade module, the following three tables are created:

- "**informix**".**mqiservice**
- "**informix**".**mqipubsub**
- "**informix**".**mqipolicy**

These tables create the definitions for services and policies and are referred to by MQ DataBlade module. Each table is explained below.

The "informix".mqiservice Table

The "**informix**".**mqiservice** table creates 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 ' ',
    PRIMARY KEY (servicename) );
```

The attributes are defined as follows:

servicename is the service name used in the MQ functions.

queuemanager is the queue manager service provider.

queuename is the queue name to send the message to or receive the message from.

defaultformat defines the default format.

ccsid is the coded character set identifier of the destination application.

The "informix".mqipubsub Table

The "**informix**".**mqipubsub** table creates 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 is the service name of the publish/subscribe service.

receiver is the queue on which to receive messages after subscription.

psstream is the stream coordinating the publish/subscribe service.

pubsubtype is the service type.

The "informix".mqipolicy Table

The "**informix**".**mqipolicy** table creates 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')),
    PRIMARY KEY (policyname) );
```

The attributes are defined as follows:

policyname
: is the name of the policy.

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

snd_retrycount
is 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
is the interval (in milliseconds) between each retry.

snd_newcorrelid
is 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
is 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
is 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
is 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
is whether Exception reports are required.

snd_rt_coa
is whether Confirm on Arrival reports are required.

snd_rt_cod
is whether Confirm on Delivery reports are required.

snd_rt_expiry
is whether Expiry reports are required.

reply_q is the name of the reply queue.

reply_qmgr
is the name of the reply Queue Manager.

rcv_truncatedmsg
is whether truncated messages are accepted.

rcv_convert
: is whether the message is code page converted by the message transport when received.

rcv_poisonmsg
: is 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
: is whether the queue is opened as a shared queue.

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

pub_suppressreg
: is whether implicit registration of the publisher is suppressed. (This attribute is ignored for WebSphere MQ Integrator Version 2.)

pub_anonymous
: is whether the publisher registers anonymously.

pub_publocal
: is whether the publication is only sent to subscribers that are local to the broker.

pub_direct
: is whether the publisher should accept direct requests from subscribers.

pub_correlasid
: is whether the Correl ID is used by the broker as part of the publisher's identity.

pub_retain
: is whether the publication is retained by the broker.

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

sub_anonymous
: is whether the subscriber registers anonymously.

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

sub_newpubsonly
: is whether the subscriber is not sent existing retained publications when it registers.

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

sub_correlasid
: is the broker as part of the subscriber's identity.

sub_informifret
: is whether the broker informs the subscriber if a publication is retained.

sub_unsuball
is whether all topics for this subscriber are to be deregistered.

Default Table Values

Most of the MQ DataBlade functions have an optional *policy* and *service* parameter. If the parameter is not passed, the default value is used. Table 6-1 lists the default value that MQ DataBlade module will use.

Table 6-1. Default Values

Type	Name	Resources	Notes
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, as listed in Table 6-1 above. 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 of the MQ DataBlade module, 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');
```

Registering the MQ DataBlade Module

The MQ DataBlade module is distributed with Informix Dynamic Server Versions 10.00.xC3 and above. Use BladeManager to register the MQ DataBlade module, 2.0 in each database from which your want to access WMQ. See the *IBM Informix DataBlade Module Installation and Registration Guide* for more information.

Note: Information on how to unregister your DataBlade is also available in the *IBM Informix DataBlade Module Installation and Registration Guide*. During unregistration, the "**informix**"**.mqi*** tables are dropped. You must unload the table data and re-load the tables again.

Verification

This section includes sample code to enable you to verify MQ DataBlade module functionality. For more information about all of the functions used below, see Chapter 8, "MQ DataBlade Functions."

The following actions are described in this section:

- Inserting data into a queue
- Reading an entry from a queue
- Receiving an entry from a queue
- Publishing and subscribing to a queue

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

Inserting 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;
```

Reading 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;
```

Receiving an Entry from a Queue

The **MQReceive()** function removes the message from the queue, as in the following example:

```
BEGIN WORK;

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

(expression)  hello queue

1 row(s) retrieved.

COMMIT WORK;
```

Publishing and Subscribing to a Queue

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

The following actions are described in this section:

- Subscribing to a queue
- Unsubscribing from a queue
- Publishing to a queue

Subscribing 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;
```

Unsubscribing From a Queue

To unsubscribe from a queue, use the **MQUnsubscribe()** function, as in the following example:

```
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;
```

Publishing to a Queue

To publish to a queue, use the **MQPublish()** function, as in the following example:

```
BEGIN WORK;

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

(expression)           1

COMMIT WORK;
```

Chapter 7. MQ DataBlade Tables

In This Chapter

This chapter describes how to map WMQ queues to Dynamic Server relational tables and how to use those tables.

MQ DataBlade Tables

The MQ DataBlade module provides the IBM Informix Virtual-Table Interface (VTI) access method to access WMQ queues using Dynamic Server 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 on VTI, see the *IBM Informix Virtual-Table Interface Programmer's Guide*.

Working with Tables

This section describes how to work with tables:

- Schema mapping
- General table behavior
- Creating and binding a table
- Using INSERT and SELECT statements
- Retrieving the queue element
- Special considerations
- Table errors

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 7-1. 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

Table 7-1. Schema Mapping to WMQ Objects (continued)

Name	Type	Description
msgid	varchar(12)	The message ID
msgformat	varchar(8)	The message format

General Table Behavior

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.

Creating and Binding 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 on the **MQCreateVtiRead()** or **MQCreateVtiReceive()** functions, see Chapter 8, "MQ DataBlade Functions."

Using 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
```

Retrieving the Queue Element

Use the **MQRead()** function to retrieve the queue element:

```
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
```

Chapter 8. MQ DataBlade Functions

In This Chapter

This chapter describes each MQ DataBlade function and provides detailed information about each function's syntax and usage.

MQ DataBlade Functions Overview

The MQ DataBlade module provides functions to enable Dynamic Server applications to exchange data directly between the application and Websphere MQ. All MQ DataBlade functions are created with a stacksize of 64K. These MQ DataBlade functions can be executed within SQL statements and should have an explicit or implicit transactional context. Examples of SQL code using MQ DataBlade functions are included in the next section.

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

MQ DataBlade Functions

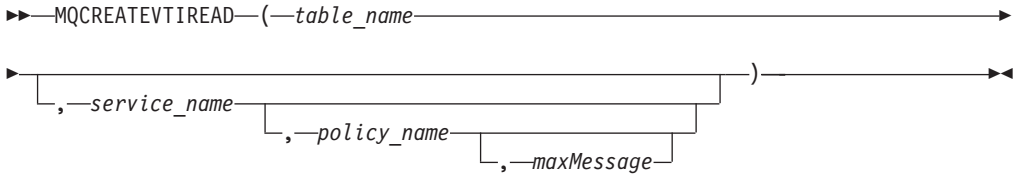
When you register the MQ DataBlade module in a database, the following functions become available:

- MQCreateVtiRead()
- MQCreateVtiReceive()
- MQPublish()
- MQRead()
- MQReadClob()
- MQReceive()
- MQReceiveClob()
- MQSend()
- MQSendClob()
- MQSubscribe()
- MQTrace()
- MQUnsubscribe()
- MQVersion()

MQCreateVtiRead()

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

Syntax



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 8-1. 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');
```

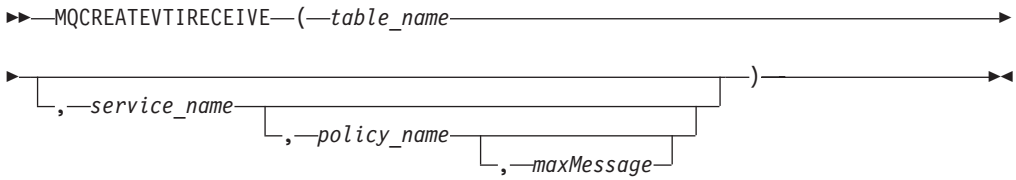
Read a message from the queue:

```
select * from VtiReadTest;
```

MQCreateVtiReceive()

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

Syntax



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 8-2. 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:

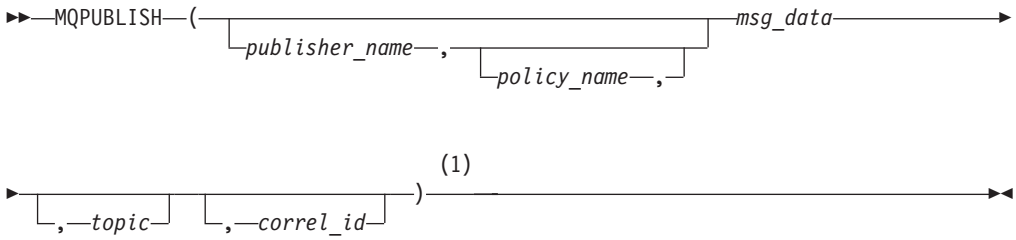
```
select * from VtiReceiveTest;
```

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.

MQPublish()

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

Syntax



Notes:

1 See the Usage section for argument interpretation.

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 8-3. 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.

	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;
```

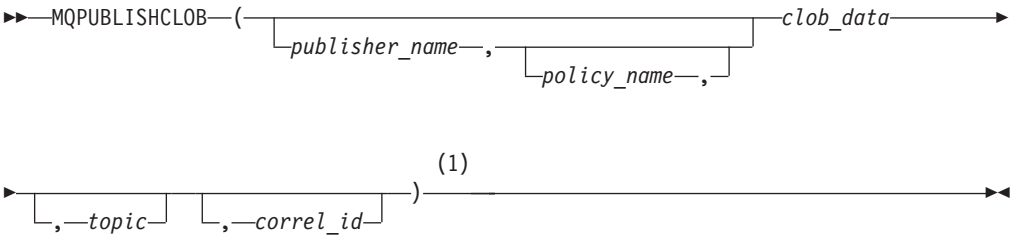
This statement receives the message with the following parameters (it returns "Rain"):

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

MQPublishClob()

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

Syntax



Notes:

1 See the Usage section for argument interpretation.

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 8-4. 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* arg2 = *policy_name* arg3 = *clob_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 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.

	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;
```

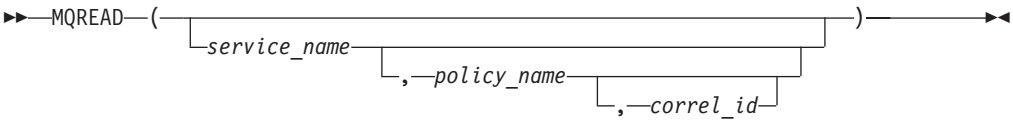
This statement receives the message with the following parameters:

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

MQRead()

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

Syntax



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 8-5. MQRead() argument interpretation

Usage	Argument Interpretation
MQRead()	No arguments
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'));
```

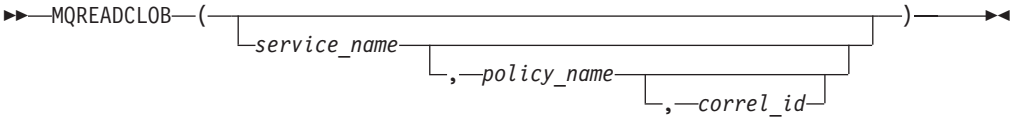
This example reads the message at the head of the queue with the following parameters:

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

MQReadClob()

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

Syntax



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 8-6. 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.

Examples

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'));
```

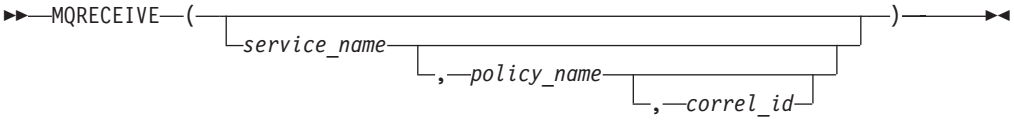
This example reads the content of the message as a CLOB at the head of the queue into the CLOB with the following parameters:

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

MQReceive()

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

Syntax



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 8-7. 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'));
```

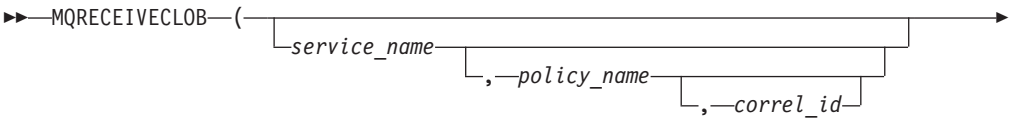
This example receives the message at the head of the queue with the following parameters:

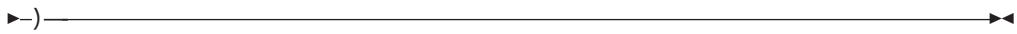
- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *correl_id*: "1234"

MQReceiveClob()

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

Syntax





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 8-8. 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* 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'));
```

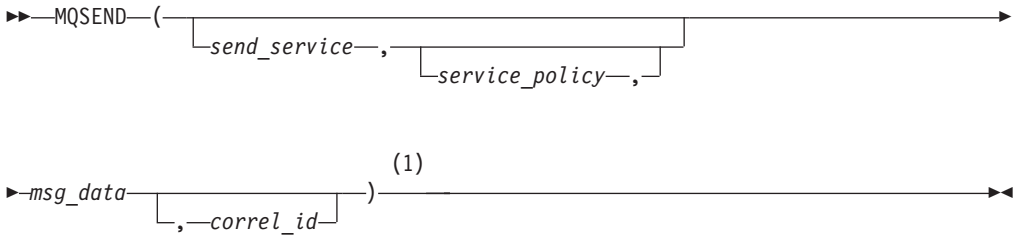
This example receives the content of the message as a CLOB at the head of the queue into the CLOB with the following parameters:

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

MQSend()

The **MQSend()** function puts the message into the WMQ queue.

Syntax



Notes:

1 See the Usage section for information on argument interpretation.

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 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 **MQSend()** function are interpreted.

Table 8-9. 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;
```

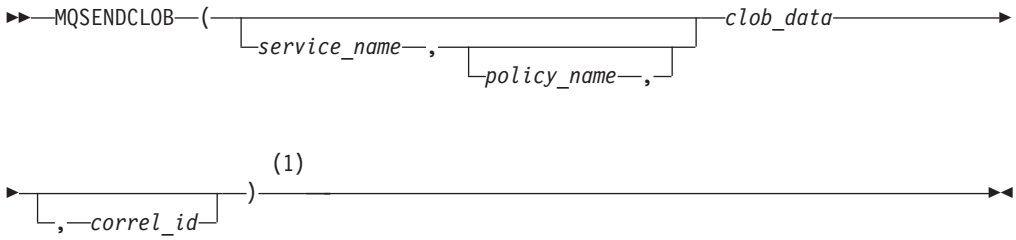
This example sends the message to the WMQ with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *msg_data*: "Testing 678"
- *correl_id*: "TEST3"

MQSendClob()

The **MQSendClob()** function puts the CLOB data into the WMQ queue.

Syntax



Notes:

1 See the Usage section for information on argument interpretation.

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 *send_service*, 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 8-10. 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
- *msg_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;
```

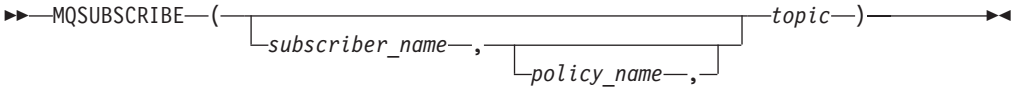
This example sends a CLOB to the WMQ with the following parameters:

- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *msg_data*: filetoclob("/work/mydata", "client")
- *correl_id*: "TEST3"

MQSubscribe()

The **MQSubscribe()** function is used to register interest in WMQ messages published on one or more topics.

Syntax



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 8-11. 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.

	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;
```

The above 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;
```

The above 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;
```

The above 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;
```

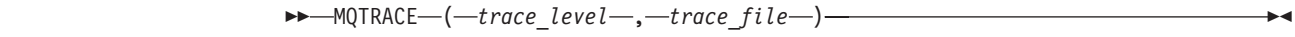
This example demonstrates a subscriber registering an interest in messages containing the topics "Stocks" and "Bonds" with the following parameters:

- *subscriber_name*: "PORTFOLIO-UPDATES"
- *policy_name*: "BASIC-POLICY"
- *topic*: "Stocks", "Bonds"

MQTrace()

The **MQTrace()** procedure specifies the level of tracing and the location to which the trace file is written.

Syntax



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 *IBM Informix Guide to SQL: Reference*.

Examples

Example 1:

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:

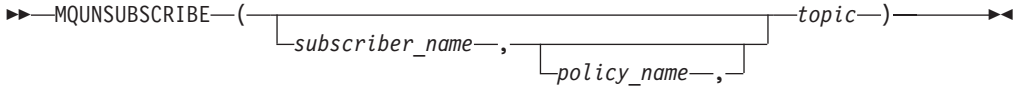
```
14:19:38 Trace ON level : 50
14:19:47  >>ENTER : mqSend<<
14:19:47    status:corrid is null
14:19:47  >>ENTER : MqOpen<<
14:19:47   status:MqOpen @ build_get_mq_cache()
14:19:47  >>ENTER : build_get_mq_cache<<
14:19:47   status:build_get_mq_cache @ mi_get_database_info()
14:19:47   status:build_get_mq_cache @ 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()

The **MQUnsubscribe()** function is used to unregister interest in WMQ messages published on one or more topics.

Syntax



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 8-12. MQUnsubscribe() argument interpretation

Usage	Argument Interpretation
MQUnsubscribe(arg1)	arg1 = *topic*
MQUnsubscribe(arg1, arg2)	arg1 = *service_name* arg2 = *topic*

Table 8-12. MQUnsubscribe() argument interpretation (continued)

Usage	Argument Interpretation
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()

The **MQVersion()** function returns the version of the MQ DataBlade module.

Syntax

Example

Show the version:

```
EXECUTE FUNCTION MQVersion();
OutPut of the MQVersion() function: MQBLADE 2.0 on 29-MAR-2005
```

Chapter 9. MQ DataBlade Module Error Handling

In This Chapter

This chapter provides information on MQ DataBlade error codes.

Error Codes

This section describes MQ DataBlade 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%
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""

Part 3. Binary Data Types

Chapter 10. About the Binary DataBlade Module

In This Chapter

This chapter provides an overview of the Binary DataBlade module and lists its requirements and limitations.

Binary DataBlade Module Overview

The Binary DataBlade module includes the binary18 and binaryvar data types that allow you to store binary-encoded strings, which can be indexed for quick retrieval. The Binary DataBlade module comes with 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, indexing, and storage are discussed in Chapter 11, "Storing and Indexing Binary Data," on page 11-1 Binary DataBlade functions are discussed in Chapter 12, "Binary DataBlade Functions," on page 12-1

In addition, the Binary DataBlade module has the following features:

- The binary data types are allowed in Enterprise Replication.
- Casts to and from the LVARCHAR data type are permitted as well as implicit casts between the binary18 and binaryvar data types.
- The aggregate functions **COUNT DISTINCT()**, **DISTINCT()**, **MAX()**, and **MIN()** are supported.

Binary DataBlade Module Limitations

The Binary DataBlade module has the following limitations:

- The only arithmetic operations supported are the bitwise operators included in this DataBlade module: **bit_and()**, **bit_or()**, **bit_xor()**, and **bit_complement()**.
- The LIKE and MATCHES conditions are not supported.

Software Requirements

The Binary DataBlade module is supported on IBM Informix Dynamic Server, Version 10.00.xC6 or later.

Registering the Binary DataBlade Module

Use BladeManager to register the Binary DataBlade module in each database in which you want to use binary user-defined types and functions. See the *IBM Informix DataBlade Module Installation and Registration Guide* for more information.

You cannot register the Binary DataBlade module into an ANSI-compliant database.

Unregistering the Binary DataBlade Module

Unregistration removes the definitions of the Binary DataBlade module's user-defined binary data types and their associated functions. You can unregister a DataBlade module that was previously installed only if there is no data in your database that uses the definitions defined by the DataBlade module. This means, for example, that if a table currently uses the binaryvar or the binary18 data type, you cannot unregister the Binary DataBlade module from your database. Refer to the *IBM Informix DataBlade Module Installation and Registration Guide* for instructions on how to unregister DataBlade modules.

Chapter 11. Storing and Indexing Binary Data

In This Chapter

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.

binaryvar Data Type

The binaryvar data type is a variable-length opaque type with a maximum length of 255 bytes.

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 case-insensitive 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')
```

Inserting 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;
}
```

Indexing Binary Data

The binaryvar and binary18 data types support indexing using the B-tree access method for single-column indexes as well as 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;
```

Chapter 12. Binary DataBlade Functions

In This Chapter

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.

bit_and()

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 Values

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
```

bit_complement()

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 Values

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
```

bit_or()

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 Values

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
```

bit_xor()

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 Values

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
```

Supporting 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 **bdtrelease()** function returns the release version number of the Binary DataBlade module. The **bdttrace()** function is used to trace events related to the Binary Data Type module.

bdtrelease()

The **bdtrelease()** function provides the release version number of the Binary DataBlade module.

Syntax

```
bdtrelease(void)
```

Usage

Use the **bdtrelease()** function when directed to do so by an IBM technical support representative.

Return codes

This function returns the name and release version number of the Binary DataBlade module

Example

Example output:

```
execute function bdtrelease();

(expression)  BinaryString DataBlade Release 1.0a Patch level 0 (Build 107)
              Compiled on Tue Apr 17 13:49:40 EDT 2007 with:
                 IBM Informix Dynamic Server Version 11.10.FC1
                 glslib-4.50.UC1_B1
```

bdttrace()

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 **$INFORMIXDIR/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 the Binary DataBlade Module.

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 *IBM Informix Guide to SQL: Reference*.

Example

```
bdttrace(tracefile)
```

LENGTH()

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 *IBM Informix Guide to SQL: Reference*.

Example

```
SELECT length(binaryvar_col) FROM table WHERE binaryvar_col = '0A010204'
expression
------------------
 4
```

OCTET_LENGTH()

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 *IBM Informix Guide to SQL: Reference*.

Example

```
SELECT octet_length(binaryvar_col) FROM table WHERE binaryvar_col = '93FB'
expression
------------------
 2
```

Part 4. Basic Text Search

Chapter 13. About the Basic Text Search DataBlade Module

In This Chapter

This chapter gives an overview of the Basic Text Search (BTS) DataBlade module and provides the steps you need to prepare the module such as creating a **bts** index before you can use the search feature.

Overview of the Basic Text Search DataBlade Module

The Basic Text Search DataBlade module allows you to search 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. The Basic Text Search DataBlade module uses the open source CLucene text search package. 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 one of the database server-controlled virtual processes.

The Basic Text Search DataBlade module has two principal components, the **bts_contains()** search predicate function and the Basic Text Search DataBlade management functions.

The bts_contains() Search Predicate

When you execute searches with Basic Text Search 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.

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.

For the complete syntax and examples illustrating the use of the **bts_contains()** search predicate, see Chapter 14, "Basic Text Search Queries," on page 14-1.

Basic Text Search DataBlade Module Functions

The Basic Text Search DataBlade module includes several functions that you can use to perform tasks, such as compacting the **bts** index and obtaining the release number of the module. For a complete list of Basic Text Search functions and their usage, see Chapter 16, "Basic Text Search Functions," on page 16-1.

Requirements and Restrictions for the Basic Text Search DataBlade Module

The following sections list the requirements and restrictions for the Basic Text Search DataBlade module.

For restrictions that apply to bts queries, see "Basic Text Search Query Restrictions" on page 14-1.

Database Server Requirements and Restrictions

If you want to use the Basic Text Search DataBlade module, you must have IBM Informix Dynamic Server, Version, 11.10, or a later version.

Basic Text Search has the following restrictions:

- Basic Text Search can be used with most multi-byte character sets and supports GLS, including UTF-8. The exception is ideographic languages such as Chinese, Korean, and Japanese.
- Basic Text Search does not support:
  - Distributed queries
  - Parallel Database Queries (PDQs)

Basic Text Search supports searches on primary and all types of secondary servers in high-availability clusters.

Supported Data Types for Basic Text Search

To use Basic Text Search, you must store the text data in a column of data type BLOB, CHAR, CLOB, LVARCHAR, NCHAR, NVARCHAR, or VARCHAR. The index can be stored in either an sbspace or an extspace.

Note: Although you can store searchable text in a column of the BLOB data type, Basic Text Search does not support indexing binary data. BLOB data type columns must contain text.

If your documents are over 32 KB, store them in columns of type BLOB or CLOB.

Index Restrictions for Basic Text Search

The following characteristics are **not** supported for **bts** indexes:

- Composite indexes
- Fill factors
- Index clustering
- Unique indexes

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.

Preparing the Basic Text Search DataBlade Module

Before you can use the Basic Text Search DataBlade module, you must prepare the server environment and create the **bts** index.

Prerequisites

- Review the "Database Server Requirements and Restrictions" on page 13-2.
- Verify the searchable text is one of the "Supported Data Types for Basic Text Search" on page 13-2.
- Review "Index Restrictions for Basic Text Search."

Procedure

To prepare the Basic Text Search DataBlade module, complete these tasks:

1. Define the **bts** extension virtual process class.
2. Create a default sbspace.
3. Create an sbspace for the **bts** index.
4. Optional. Create a space for temporary data.
5. Create the **bts** index.

Defining the bts Extension Virtual Processor Class

You must define a **bts** extension virtual processor (EVP) class to use a **bts** index. The Basic Text Search functions run in the **bts** EVP without yielding, which means that only one index operation executes at one time. Basic Text Search supports only 1 **bts** EVP.

To define a **bts** virtual processor, add the following line to the ONCONFIG file:

```
VPCLASS bts,noyield,num=1
```

Restart the database server for the **bts** processor class to take effect.

For information about virtual processors, see the *IBM Informix Administrator's Guide*.

Creating a Default sbspace

You must create a default sbspace and set the SBSPACENAME configuration parameter in the ONCONFIG file before you register the DataBlade module into any database, or the registration fails. During registration, the Basic Text Search DataBlade module sets up internal directories in a default sbspace.

The Basic Text Search DataBlade module 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.

The default sbspace must have these characteristics:

- Logging must be enabled. Include the **-Df "LOGGING=ON"** option when you create the sbspace with the **onspaces** utility.
- Buffering must be enabled. Buffering is enabled by default when you create an sbspace with the **onspaces** utility.

To create the default sbspace:

1. Set the SBSPACENAME configuration parameter in the ONCONFIG file to the name of your default sbspace.

   The following example sets the name of the default sbspace to **sbsp1**:

   ```
   SBSPACENAME sbsp1
   ```

2. Restart the database server.
3. Create the sbspace by using the **onspaces** utility. Include the **-Df "LOGGING=ON"** option.

   The following example creates an sbspace called sbsp1 in the file **c:/IFMXDATA/sbspace**:

   ```
   onspaces -c -S sbsp1 -g 2 -p c:/IFMXDATA/sbspace -o 0 -s 100000 -Df "LOGGING=ON"
   ```

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.

If you do not create a separate sbspace for your **bts** indexes, the Basic Text Search DataBlade module 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 your **bts** index must have these characteristics:

- Logging must be enabled. Include the **-Df "LOGGING=ON"** option when you create the sbspace with the **onspaces** utility.
- Buffering must be enabled. Buffering is enabled by default when you create an sbspace with the **onspaces** utility.

To create an sbspace, use the **onspaces** utility. For example:

```
onspaces -c -S bts_sbspace -o 0 -s 100000 -p /dev/sbspace -Df LOGGING=ON
```

To create an extspace:

1. Create a directory for the index.
2. Create the extspace by using the **onspaces** utility. Include the **-Df "LOGGING=ON"** option.

The following example creates a directory and an extspace:

```
mkdir bts_extspace_directory
onspaces -c -x bts_extspace -l "/bts_extspace_directory"
```

For information about the **onspaces** utility and creating a space, see ″The onspaces Utility″ in the *IBM Informix Administrator's Reference*.

Creating a Space for Temporary Data

The Basic Text Search DataBlade module 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. For best performance, the space should be a temporary sbspace. However, you can also use an sbspace or an extspace. The temporary space should not have logging enabled.

If you do not specify a separate sbspace for temporary data when you create the **bts** index, where the Basic Text Search DataBlade module stores temporary data in the first of the following locations that is defined:

- The sbspace specified by the SBSPACETEMP configuration parameter.
- The sbspace specified in the CREATE INDEX statement.
- The sbspace specified by the SBSPACENAME configuration parameter.

To create a temporary sbspace, use the **onspaces** utility with the **-t** option. (Do not include the **-Df "LOGGING=ON"** option.) For example:

```
onspaces -c -S temp_sbspace -t -o 0 -s 50000 -p /dev/temp_sbspace
```

Data and metadata for temporary files are not logged.

For information about the **onspaces** utility and creating a space, see ″The onspaces Utility″ in the *IBM Informix Administrator's Reference*.

Registering the Basic Text Search DataBlade Module

Use BladeManager to register the Basic Text Search DataBlade module in each database in which you want to use it. See the *IBM Informix DataBlade Module Installation and Registration Guide* for more information.

Note: Registration fails if the SBSPACENAME configuration parameter is not set in the ONCONFIG file.

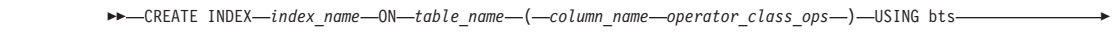
Creating the Index by Specifying the bts Access Method

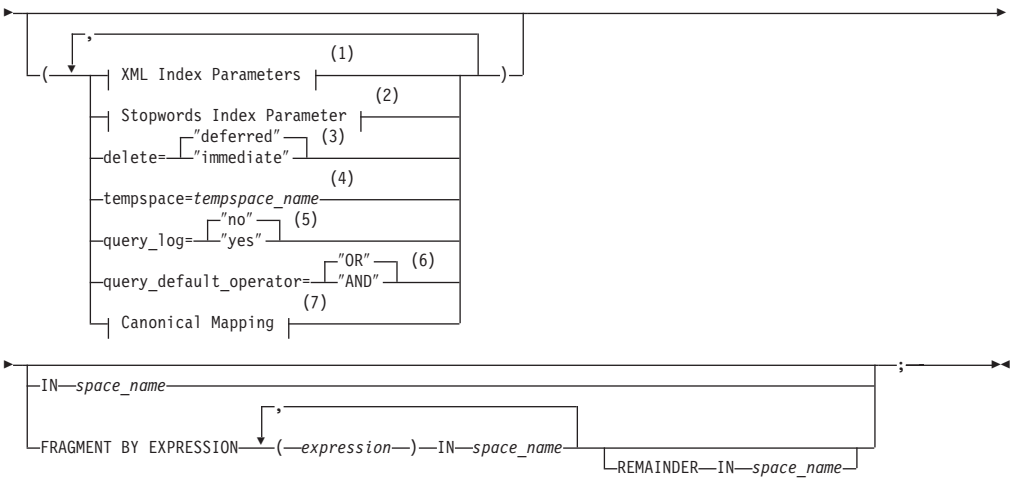
You must create a **bts** index for each text column that you plan to search.

The **bts** access method is a secondary access method that allows you to call on the Basic Text Search engine to create indexes that support Basic Text Search.

You cannot alter the characteristics of a **bts** index after you create it. Instead, you must drop the index and re-create it with the desired characteristics.

To create a **bts** index, use the following syntax:





Notes:

1 See "Basic Text Search XML Index Parameters Syntax" on page 15-2.

2 See "Customized Stopword List" on page 14-7.

3 See "Optimizing the bts Index" on page 17-1.

4 See "Creating a Space for Temporary Data" on page 13-5.

5 See "Tracking Queries on bts Indexes" on page 13-7.

6 See "Boolean Operators" on page 14-5.

7 See "Syntax of canonical_maps Index Parameter" on page 14-9.

.

expression Expression defining an index fragment. Must return a Boolean value. Can contain only columns from the current table and data values from only a single row. No subqueries nor aggregates are allowed. The built-in CURRENT, DATE, SYSDATE, and TODAY functions are not valid here. The **bts_contains()** search predicate is not valid. For more information on expressions, see *IBM Informix Guide to SQL: Syntax*.

index_name The name of the **bts** index.

table_name The name of the table for which you are creating the index.

column_name The name of the column in the table that contains the text documents to search.

operator_class_ops
The operator class applicable to the data type specified in the *column_name*. Refer to "The Operator Class" on page 13-7 for valid operator class names and their corresponding data types.

tempspace_name
The name of the space in which to store temporary files.

space_name The directory specified as the sbspace or extspace in which to store the **bts** index.

For example, suppose your search data is contained in a column **brands**, of data type CHAR, in a **products** table. To create a **bts** index 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;
```

The following example stores the **bts** 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;
```

The Operator Class

When you create a **bts** index, you must specify the operator class defined for the data type of the column being indexed. An operator class is a set of functions that Informix Dynamic 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 13-1. Data Types and Their Corresponding Operator Classes

Data Type	Operator Class
BLOB	bts_blob_ops
CHAR	bts_char_ops
CLOB	bts_clob_ops
LVARCHAR	bts_lvarchar_ops
NCHAR	bts_nchar_ops
NVARCHAR	bts_nvarchar_ops
VARCHAR	bts_varchar_ops

Tracking Queries on bts Indexes

You can determine the frequency of queries that are run against a **bts** index by enabling tracking.

When tracking is enabled, each query run against the **bts** index produces a log record in the **$INFORMIXDIR/tmp/bts_query.log** file. Each log record has five fields, separated by a pipe character (|):

query time stamp|*index name*|*partn*|*query*|*number of rows*|

The fields are described in the following table.

Table 13-2. 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.

Table 13-2. Query tracking fields (continued)

Field Name	Data Type	Description
Number of rows	INTEGER	The number of rows returned by the query.

You can view the log records by loading them into a table and then querying the table.

To track queries: ,

1. Include the **query_log="yes"** parameter in the CREATE INDEX statement when you create your **bts** index.
2. Create a table to hold the log records.
3. Load the log records into the log table.
4. Query the log table to view the records.

The following example shows how to perform each of these steps.

Create the **bts** index with tracking enabled:

```
CREATE INDEX bts_idx ON products (brands  bts_char_ops)
 USING bts (query_log="yes") IN sbsp1;
```

Create a table to hold the log records:

```
CREATE TABLE bts_query_log_data(
  qwhen DATETIME YEAR TO FRACTION,
  idx_name LVARCHAR,
  partn INTEGER,
  query LVARCHAR,
  rows INTEGER);
```

Load the log records into the table:

```
LOAD FROM '$INFORMIXDIR/tmp/bts_query.log' INSERT INTO bts_query_log_data;
```

Query the table to view the log records:

```
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 austin
rows 3

3 row(s) retrieved.
```

Chapter 14. Basic Text Search Queries

In This Chapter

This chapter shows the syntax for the **bts_contains()** search predicate and discusses the different types of searches you can perform with the Basic Text Search DataBlade module.

Searching with Basic Text Search

The Basic Text Search module supports many types of searches, such as word, phrase, Boolean, proximity, and fuzzy. Searches are performed using the **bts_contains()** search predicate. Before you can perform a search, you must create a **bts** index on the column you want to search.

For information about creating a **bts** index, see "Creating the Index by Specifying the bts Access Method" on page 13-5.

Basic Text Search Query Restrictions

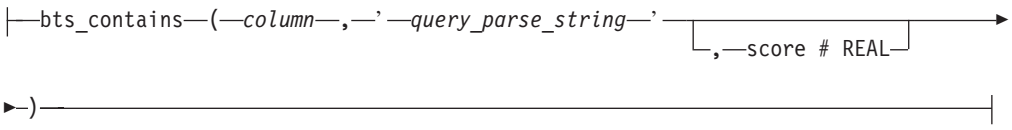
Basic Text Search queries have the following restrictions:

- Searches are not case-sensitive.
- The SQL Boolean predicates AND, OR, and NOT cannot be used between **bts_contains()** search predicates. For example the expression, `bts_contains(column, 'word1') AND bts_contains(column, 'word2')` is not supported. However, the expression, `bts_contains(column, 'word1 AND word2')` is correct, where the Boolean operator (`AND`) is within the search predicate.

Basic Text Search Query Syntax

The **bts_contains()** search predicate has the following syntax:

bts_contains() Search Predicate:



column The column to be searched. It must be a single column for which a **bts** index has been defined.

query_parse_string
The word or phrase that is being searched as well as optional search operators. Enclose the *query_parse_string* within single quotation marks. If the data is indexed with XML index parameters, include the XML tag field or path field followed by the searchable text in the format *fieldname*:*string*. For information about indexing and searching XML data, see Chapter 15, "Basic Text Search XML Index Parameters," on page 15-1.

score # REAL Optional argument 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 the results to the search criteria, compared to that of other indexed records. The higher the document score value, the more closely the results match 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 as in the following example, which returns the word `standard` in the column **brands** in a table called **products** only when the document score value is greater than 70.

```
SELECT id FROM products
WHERE bts_contains(brands, 'standard', score # REAL)
AND score > 70.0;
```

For more information about SLVs, see the *IBM Informix Guide to SQL: Syntax*.

Basic Text Search Query Types

The following sections provide the syntax and examples of the different types of Basic Text searches.

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 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 spaces and punctuation 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 lower case according 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>" ')
```

Basic Text Search Fields

The Basic Text Search module 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 a field in the **bts_contains()** search predicate because the default field contents is always searched. However, when you index structured text by using 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 tags 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*. For example if the XML data is indexed in a field 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:citrus namespace:

```
bts_contains(column, ' fruit\:citrus:Orange ')
```

For information about indexing and searching for XML data, see Chapter 15, "Basic Text Search XML Index Parameters," on page 15-1.

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

Basic Text Search supports single-character and multiple-character wildcard searches.

Single-Character Wildcard Searches: 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')
```

Multiple-Character Wildcard Searches: To perform a multiple-character wildcard search, use an asterisk (*) in the search term. Multiple-character wildcard searches look for zero or more characters. 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 search for `contour`, `crater`, and `color`:

```
bts_contains(column, 'c*r')
```

You can use a single wildcard character (?) as the first character of the search term. You cannot use a multiple wildcard character (*) as the first character of the search term.

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. For example:

```
bts_contains(column, 'bank~0.9')
```

The default degree of similarity is 0.5.

Proximity Searches

A proximity search allows you to specify the number of nonsearch words that can occur between search terms. 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 within 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).

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} ')
```

Boosting a Term

By default, all terms have equal value when sorted in the search results. Boosting a term assigns more relevance to a word or phrase. The search results are the same, but the specified term appears 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. 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  ')
```

By default the boost factor is 1. It must be a positive integer, but it can be less than one. For example .3 or .5.

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 "Creating the Index by Specifying the bts Access Method" on page 13-5.

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 word `travel` in the `book` field and the phrase `john stewart` in the `author` field:

```
bts_contains(column, ' book:travel AND author:"john stewart" ')
```

The Required Operator (+): The required operator, which is denoted by the plus sign (+) means that the term following it must exist in the document. When you use the required operator before every term, it has the same functionality as the AND operator. For example this search predicate finds documents that contain both the word `UNIX` and the phrase `operating system`

```
bts_contains(column, ' +UNIX  +"operating system"  ')
```

However, if you use the required operator before only one of the terms, it means that string must appear in the document and the other term *might* appear. To search for documents that must contain the term `UNIX` and might contain the term `Windows`, use this search predicate:

```
bts_contains(column, ' +UNIX  Windows ')
```

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 dash (-).

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 ')
```

Grouping Words and Phrases

You can group words and phrases in parentheses to form more complex queries using 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 will return 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.

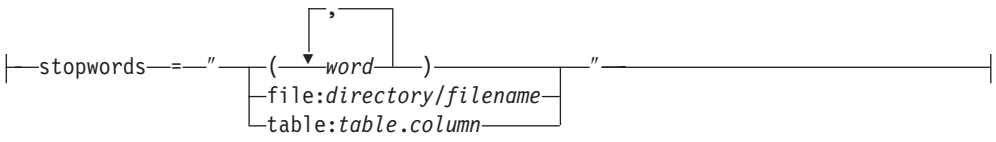
Basic Text Search Stopwords

Stopwords are excluded from the **bts** index and are not searchable. Stopwords can reduce the time it takes to perform a search, reduce index size, and help avoid false results. You can create a customized stopword list for frequently occurring words in your data or you can use the default stopword list.

Customized Stopword List

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. For the complete syntax see "Creating the Index by Specifying the bts Access Method" on page 13-5. Following is a syntax segment for the **stopwords** index parameter:

Stopwords Index Parameter:



column The name of the column containing stopwords.

directory
The directory path for the stopwords file.

filename
The name of the file containing stopwords.

table The name of the table containing the column with stopwords.

word The term to use as a stopword. Stopwords must be lowercase characters.

Input for the stopwords can be one of three forms:

- inline comma-separated words
- an external file
- a table column

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 file or table 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 stopwords to the index, you must drop and re-create the index. Separate the stopwords in the file or table by commas, whitespaces, newlines, or a combination of those separators. For example:

```
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;
```

Default Basic Text Search Stopword List

If you do not create a customized stopword list, the default list is used. The following words and letters are in the default stopword list for the Basic Text Search DataBlade module:

```
a an and are as at be but by for if in into is it no not of on
or s such t that the their then there these they this to was
will with
```

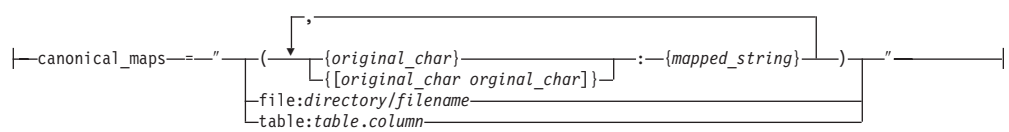
Canonical Mapping

You can map characters in your data to other characters for indexing. For example, you can specify that a letter with a diacritical mark is indexed without its diacritical mark. You can also normalize strings that tend to be inconsistent or delete character strings from indexed text.

Syntax of canonical_maps Index Parameter

You specify canonical mapping strings with the **canonical_maps** index parameter when you create the **bts** index. For the complete syntax see "Creating the Index by Specifying the bts Access Method" on page 13-5. Following is a syntax segment for the **canonical_maps** index parameter:

The canonical_maps Index Parameter:



column The name of the column containing canonical mapping strings.

directory
The directory path for the canonical mapping file.

filename
The name of the file containing canonical mapping strings.

table The name of the table containing the column with canonical mapping strings.

original_char
The characters to replace with a mapped string during indexing and searching.

mapped_string
The characters to which the original characters are replaced during indexing.

Usage

Use canonical maps to improve the accuracy of queries by equating characters with a canonical representation of those characters.

During indexing and searching, the Basic Text Search DataBlade module transforms all characters to lower case, therefore, any uppercase characters in the original characters must be mapped to lowercase characters in the mapping sting. For some locales, the uppercase characters of letters with diacritical marks or ligatures are considered independent characters from their lowercase equivalents. For those locales, you must map both the uppercase and the lowercase characters with diacritical marks or ligatures to the same lowercase letter. You cannot specify an uppercase letter in a mapped string.

Blank spaces are significant.

The Basic Text Search DataBlade module indexes and searches the mapped characters, therefore, when returning the results, words with the original characters are treated as if those characters are the same as their corresponding mapped characters. For example, if you map the character "ù" to the letter "u," then both

"Raùl" and "Raul" are indexed as "raul." Similarly, if you search for 'Raùl' or for 'Raul', all rows containing either "Raùl" or "Raul" are returned.

Map Single Characters: The following example maps the single character "ù" to the single character "u":

```
{ù}:{u}
```

Specify Multiple Original Character Strings: You can specify multiple original character stings in the same set of braces by enclosing them in brackets. Do not put a blank space between the characters when you use brackets or every blank space in the text will be indexed as the mapping string. The following example maps both "ù" and "ú" to the letter "u":

```
{[ùú]}:{u}
```

The following example also maps both "ù" and "ú" to the letter "u," but it uses two sets of mapping strings that are separated by a comma:

```
{ù}:{u},{ú}:{u}
```

Specify Multiple Characters in Mapping Strings: The mapping string can have multiple characters. For example, the following example maps the single "æ" character to the two letters "ae":

```
{æ}:{ae}
```

Prevent Indexing of Characters: You can prevent the indexing of characters by specifying empty braces for the mapping string. The following example prevents the indexing of the characters "'s":

```
{'s}:{}
```

If you want to prevent symbols from being indexed, consider how they are being used. For example, if you delete the backslash character (/) with the mapping {/}:{}, then the string ″/home/john/henry″ is indexed as ″homejohnhenry″.

Manage Multiple Spellings: You can use canonical mapping manage the inconsistent use of prefixes or other spellings. For example, if you want to search for the name ″McHenry″ and you know that the indexed name might be spelled as either "mchenry" or ″mc henry″, your query string would be:

```
 'mchenry OR "mc henry"'
```

Alternatively, you can map the two prefixes:

```
{mc }:{mc}
```

Note the space after the "mc" in the original characters. With this mapping, all the "mc henry" names are indexed as ″mchenry″ and you could search for `'mchenry'` or `'mc henry'`.

Examples of Canonical Mapping

The following examples show how to create a **bts** index with the **canonical_maps** parameter.

Example 1: Map characters as inline comma-separated strings

The following example shows how to create an index specifying two character substitutions:

```
CREATE INDEX docs_idx on repository
    (document_text bts_lvarchar_ops)
    USING bts
     (canonical_maps="({ü}:{u},{æ}:{ae})")
    IN mysbspace;
```

Example 2: Map characters as a file

The following example illustrates a file of character mappings. Some mapped characters have multiple original characters. This example assumes the locale en_us.8859-1, which does not designate uppercase letters with diacritical marks as uppercase. Therefore, both uppercase and lowercase versions of letters are included in the original characters.

```
{Ææ}:{ae},

{Œœ}:{oe},

{Ññ}:{ny},

{[ÀÁÂÃÄÅàáâãåâ]}:{a},

{[ÈÉÊËèéêë]}:{e},

{[ÌÍÎÏìíîï]}:{i},

{[ÒÓÔÖÕòóôöõ]}:{o},

{[ÙÚÛÜùúûü]}:{u},

{Çç}:{c},

{Øø}:{o},

{Ýý}:{y},

{ß}:{ss},

{mc }:{mc}
```

The following example shows how to create an index specifying a mapping file named **canon**:

```
CREATE INDEX docs_idx on repository
    (document_text bts_lvarchar_ops)
    USING bts
     (canonical_maps="file:/tmp/canon")
    IN mysbspace;
```

Searching for 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, ' \(7\+1\) ')
```

Chapter 15. Basic Text Search XML Index Parameters

In This Chapter

This chapter describes the Basic Text Search DataBlade XML index parameters and provides detailed examples about each parameter's usage.

Overview of Basic Text Search XML Index Parameters

You can use Basic Text Search XML index parameters to manipulate searches of XML data in different ways.

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 in a single field called `contents`. By contrast when you use XML index parameters, the XML tag and attribute values can be indexed in separate fields either by tag name, attribute name, or by path.

The **xmltags** or **all_xmltags** parameters identify the tags to index.

The **all_xmlattrs** parameter enables searches on all attributes that are contained in the XML tags or paths in a column that contains an XML document.

The **xmlpath_processing** parameter enables searches based on XML paths.

The **include_namespaces** parameter indexes XML tags that include namespaces.

The **include_subtag_text** parameter indexes tags and subtags as a unified string.

The **include_contents** parameter puts the XML data in original format into the `contents` field.

The **strip_xmltags** puts the XML data in an untagged format into the `contents` field.

For a basic example, given 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;
```

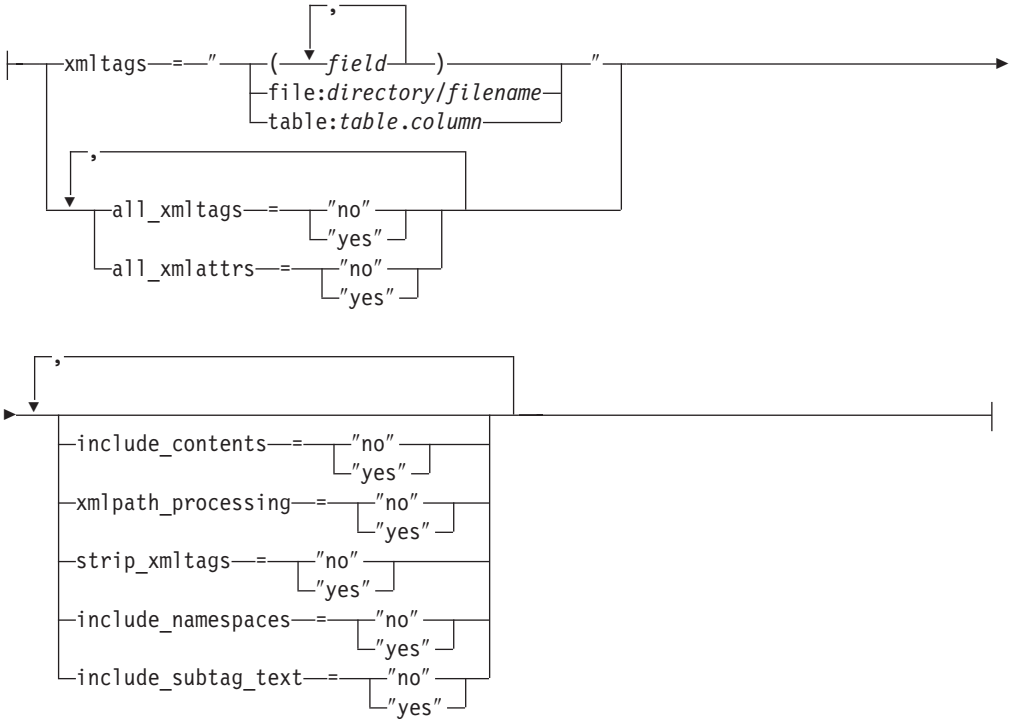
To search for a skipper's name that contains the word "Black," use the **bts** search predicate:

```
bts_contains(xml_data, 'skipper:black')
```

Basic Text Search XML Index Parameters Syntax

The Basic Text Search XML index parameters are optional parameters that you can specify when you create a **bts** index. For the complete syntax see "Creating the Index by Specifying the bts Access Method" on page 13-5. Following is a syntax segment for the XML index parameters:

XML Index Parameters:



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.

The parameters are described in the following sections:

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 **xmlpath_processing** 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, whitespaces, 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 15-7.

If you want to index all the XML tags in a column, see "The all_xmltags Index Parameter" on page 15-5.

To view the fields that you have indexed, use the **bts_index_fields()** function. See "The bts_index_fields() Function" on page 16-3.

Example: Indexing 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 15-7.

If you want to index only specific tags in a column, use the **xmltags** parameter. See "The xmltags Index Parameter" on page 15-3.

To view the fields that you have indexed, use the **bts_index_fields()** function. See "The bts_index_fields() Function" on page 16-3.

Example: Indexing 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 "The bts_index_fields() Function" on page 16-3.

Examples: Indexing 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 "The bts_index_fields() Function" on page 16-3.

Example: Indexing 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 "The bts_index_fields() Function" on page 16-3.

Example: Indexing 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 "The bts_index_fields() Function" on page 16-3.

Example: Indexing 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 "The bts_index_fields() Function" on page 16-3.

Example: Indexing 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 or italic 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 "The bts_index_fields() Function" on page 16-3.

Example: Indexing 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
```

Chapter 16. Basic Text Search Functions

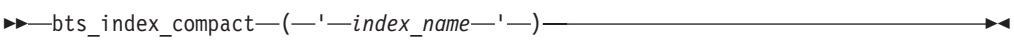
In This Chapter

This chapter describes the Basic Text Search DataBlade functions and provides detailed information about each function's syntax and usage.

The bts_index_compact() Function

The **bts_index_compact()** function deletes all documents from the **bts** index that are marked as deleted.

Syntax



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 "Optimizing the bts Index" on page 17-1.

Return codes

t The operation was successful.

f The operation was unsuccessful.

Example

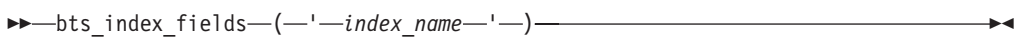
The following example compacts the **bts** index `desc_idx`:

```
EXECUTE FUNCTION bts_index_compact('desc_idx');
```

The bts_index_fields() Function

The **bts_index_fields()** function returns the list of indexed field names in the **bts** index.

Syntax



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, separated by forward slash (/) characters.

When you do not use Basic Text Search XML index parameters, the **bts_index_fields()** function returns one default field called `contents`. When you use XML index parameters, the XML data is indexed in separate fields by tag name or by path. The `contents` field is not indexed unless you also enable the **include_contents** parameter.

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 at a later time you add a row that contains the specified tag name, the field name for that tag will appear in the output.

The **bts_index_fields()** function returns the field names in alphabetical order.

Example

Given the XML fragment:

```
<boat>
 <skipper>Captain Jack</skipper>
 <boatname>Black Pearl</boatname>
</boat>
```

If you create an index without XML index parameters:

```
CREATE INDEX boats_bts ON boats(boat_data bts_lvarchar_ops)
USING bts IN bts_sbspace;
```

The **bts_index_fields()** function will return the default field:

```
contents
```

If you create an index with XML index parameters:

```
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 will return 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.

If you create an index with the **all_xmltags** and the **xmlpath_processing** parameters enabled:

```
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 will return field names that include full paths:

```
/boat/boatname
/boat/skipper
```

If you create an index with the **include_contents** parameter enabled:

```
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 will return the following fields:

```
boatname
contents
skipper
```

For information about the XML index parameters, see Chapter 15, "Basic Text Search XML Index Parameters," on page 15-1.

The bts_release() Function

The **bts_release()** function provides the release version number of the Basic Text Search DataBlade module.

Syntax

Usage

Use the **bts_release()** function if IBM Technical Support asks you for the Basic Text Search DataBlade module version number.

Return codes

This function returns the name and release version number of the Basic Text Search DataBlade module.

Example

Example output:

```
+                BTS 2.00 Compiled on Tue Mar 31 11:25:52 CDT 2009
```

The 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

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 **$INFORMIXDIR/tmp** directory.

Usage

Use the **bts_tracefile()** function to troubleshoot events related to the Basic Text Search DataBlade Module.

For the syntax for **bts_tracelevel()**, see "The bts_tracelevel() Function" on page 16-7.

For more details about tracing, see the *IBM Informix 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');
```

The 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

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 DataBlade Module.

For the syntax for **bts_tracefile()**, see "The bts_tracefile() Function" on page 16-6.

For more details about tracing, see the *IBM Informix 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
```

Chapter 17. Basic Text Search DataBlade Module Performance

In This Chapter

This chapter describes how to optimize the **bts** index and how transactions work with Basic Text Search.

Optimizing the bts Index

Optimizing (also known as *compacting*) the index removes index information for deleted documents and frees up disk space. Basic Text Search provides two ways for you to optimize the **bts** index: manually or automatically after every delete operation.

Tip: Disk space for documents that are marked as deleted in the bts index can be reclaimed when more documents are added. Optimizing the index releases all disk space for all deleted documents.

Deleting Rows From the bts Index Manually When Using Deferred Mode

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 made against **bts** columns do not return the deleted documents.

To release disk space 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 using the **oncheck** utility requires that you know the database name, table name, and the index name. Both methods have the same functionality.

Delete operations are faster in the deferred mode. The deferred mode is best for large indexes that are updated frequently. The indexes should be optimized (compacted) manually either with the **oncheck** utility or by using the **bts_index_compact()** function.

For information about the **oncheck** utility, see the *IBM Informix Dynamic Server Administrator's Reference*. For the syntax of the **bts_index_compact()** function, see "The bts_index_compact() Function" on page 16-2.

Deleting Rows From the bts Index Automatically with Immediate Mode

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 mode rewrites the index each time it deletes an index entry so it will slow down delete operations and make the index unusable for the period of time it takes to delete the entries.

For a description and the complete syntax of the CREATE INDEX statement for a **bts** index, including the deletion mode parameters, see "Creating the Index by Specifying the bts Access Method" on page 13-5.

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 files. A separate sbspace for
+ temporary files might also improve query speed.

See "Preparing the Basic Text Search DataBlade Module" on page 13-3 for the procedure to create a **bts** index. See the *IBM Informix Guide to SQL: Syntax* for instructions for the DROP INDEX statement.

Transactions with Basic Text Search

The **bts** index is located in an sbspace. INSERT, DELETE, and UPDATE operations lock the index during modifications, which prevents any other transaction from changing the index during the change. Therefore, each modification is done in a series. The operations make one attempt at a modification. If the index is locked, the operation fails.

The **bts** index works in READ COMMITTED isolation level regardless of the isolation level set in the database server. The READ COMMITTED isolation level provides access only to rows that have been committed. Uncommitted rows from other concurrent transaction are not accessible.

Improving Performance with Configuration Parameters

You can optimize the performance of text searches that use **bts** indexes by configuring some ONCONFIG tuning parameters.

- BUFFERPOOL

  The BUFFERPOOL configuration parameter defines a buffer pool for pages that correspond to each unique page size in use by your dbspaces. Use the BUFFERPOOL parameter to 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.

- RA_PAGES

  The RA_PAGES configuration parameter specifies the number of disk pages that the database server should attempt to read ahead during sequential scans of data or index records. Try setting this parameter to 64.

- RA_THRESHOLD

  The RA_THRESHOLD configuration parameter specifies the number of unprocessed pages in memory that signals the database server to perform the next read ahead. If RA_PAGES is set to 64, setting RA_THRESHOLD to 33 could improve the performance of your text searches.

- 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 not be swapped to disk. To do this, set this parameter to 1 (on).

- VPCLASS **noage**

  You can add the **noage** option when you specify the **bts** EVP to disable priority aging by the operating system if the operating system implements priority aging. The **noage** option for the VPCLASS parameter controls whether the operating system lowers the priority of database server processes as the processes run over a period of time. You can improve the performance of searches by setting this option to **noage**.

Before you make any changes to your ONCONFIG file, refer to the *IBM Informix Administrator's Reference* for more detailed information on each parameter.

Chapter 18. Basic Text Search DataBlade Module Error Codes

In This Chapter

This chapter provides information about Basic Text Search DataBlade error codes.

Error Codes

The following table lists Basic Text Search DataBlade error codes.

SQL State	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
BTS07	bts error - composite 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 specifed as noyield
BTS16	bts error - too many evps 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
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 field_token_max parameter: %s 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

SQL State	Description
BTS28	bts error - invalid value for min_merge parameter: %MIN_MERGE_PARAM% should be an integer value greater than 0
BTS29	bts error - invalid value for max_merge parameter: %MIN_MERGE_PARAM% should be an integer value greater than 0
BTS30	bts error - invalid value for merge_factor parameter: %MERGE_FACTOR_PARAM% should be an integer value greater than 0
BTS31	bts error - invalid value for noswchk parameter: %NOSWCHK_PARAM% should be either yes or no
BTS32	bts error - invalid value for noxtchk parameter: %NOXTCHK_PARAM% should be either yes or no
BTS33	bts error - invalid value for optimize_after_create parameter: %OPTIMIZE_AFTER_CREATE_PARAM% should be either yes 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%
BTS47	bts error - xml syntax error
BTS48	bts error - invalid value for strip_xmltags parameter: %STRIP_XMLTAGS_PARAM% should be either yes or no
BTS49	bts error - invalid value for all_xmltags parameter: %ALL_XMLTAGS_PARAM% should be either yes or no
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 - all_xmltags and xmltags parameters are mutually exclusive
BTS53	bts error - invalid value for include_contents parameter: %INCLUDE_CONTENTS_PARAM% should be either yes or no
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 - invalid value for include_namespaces parameter: %PARAM1% should be either yes or no
BTS61	bts error - invalid value for xmlpath_processing parameter: %PARAM1% should be either yes or no

SQL State	Description
BTS62	bts error - invalid value for include_subtag_text parameter: %PARAM1% should be either yes or no
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 termvector parameter: %PARAM1% should be either yes or no
BTS66	bts 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 - invalid value for all_xmlattrs parameter: %s should be either yes or no
BTS70	bts internal error - mi_lo_specset_sbspace() failed. File %s, line %d
BTS71	bts internal error - mi_lo_stat() failed. File %s, line %d
BTS72	bts internal error - mi_lo_stat_cspec() failed. File %s, line %d
BTS73	bts error - sbspace %s 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
BTS90	bts error - CLucene index exists and is locked
BTS91	bts error - CLucene index exists
BTS92	bts error - CLucene index does not exist
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%

The following Basic Text Search DataBlade error codes are generated in English under the SQLSTATE U00001.

bts error - The all_xmlattrs and xmltags parameters are mutually exclusive.
bts error - The copy_temp attribute can only be specified on an index in an sbspace.
bts error - The specified directory cannot contain a bts index.
bts error - Duplicate parameters, %s, were specified.
bts error - The field and mapping_string are mutually exclusive.
bts error - The field is not defined on the document or the value is not stored.
bts error - The fragment attribute must be specified on a fragmented index.
bts error - The GLS character name '%s' is not found.
bts error - The ID is out of bounds.
bts error - Incorrect canonical map[%d]: missing] in alternates in original characters specification.
bts error - Incorrect canonical map[%d]: missing %c in original character specification.
bts error - Incorrect canonical map[%d]: missing : in mapped characters specification.
bts error - Incorrect canonical map[%d]: missing %c in mapped characters specification.
bts error - Incorrect canonical map[%d]: spaces found in original characters string at %d.
bts error - Incorrect canonical map[%d]: trailing characters found.
bts error - Incorrect canonical map[%d]: zero length original character string.
bts error - Incorrect flag for the create_mode parameter: %s
bts error - Incorrect hex specification: \x%c%c
bts error - Incorrect value for the create_mode parameter: %s is too long.
bts error - Incorrect value for the create_mode parameter: %s should be a hexadecimal number.
bts error - Incorrect value for the create_mode parameter: %s should be an integer value greater than 0.
bts error - Incorrect value for the create_set_size parameter: %s should be an integer value greater than 0.
bts error - Incorrect value for the htr parameter: %s should be either yes or no.
bts error - Incorrect value for the insert_set_size parameter: %s should be an integer value greater than 0.
bts error - Incorrect value for the max_clause_count parameter: %s should be an integer value greater than 0.
bts error - Incorrect value for the nonorms parameter: %s should be either yes or no.
bts error - Incorrect value for the query_batch_size parameter: %s should be an integer value greater than 0.
bts error - Incorrect value for the query_default_operator parameter: %s should be either "and" or "or".
bts error - Incorrect value for the query_limit parameter: %s should be "unlimited" or an integer value greater than 0.
bts error - Incorrect value for the query_log parameter: %s should be either yes or no.
bts error - Incorrect value for the query_set_size parameter: %s should be an integer value greater than 0.
bts error - Incorrect value for the strip_nul_chars parameter: %s should be either yes or no.

bts error - Incorrect value for the tempspace parameter: %s should be an existing extspace or sbspace.
bts error - Incorrect value for the termvector parameter: %s is too long.
bts error - Incorrect value for the termvector parameter: %s should be either yes, with_positions, with_offsets or no.
bts error - Missing a comma (,) between parameters.
bts error - Missing the column name in table:%s. Use the form table:table_name.column_name.
bts error - Missing the closing parenthesis,), in a string that has an opening parenthesis: (.
bts error - Missing the closing parenthesis:). The parameter should be in the form of name=(values).
bts error - Missing a double quotation mark: ″. The parameter %s should be in the form of name=″value″.
bts error - The parameter %s should be in the form of name=value.
bts error - query attribute must be specified.
bts error - Recursive params parameter.
bts error - %s is an uppercase character. Uppercase characters are not allowed in canonical maps.
bts error - Unknown parameter name: %.*s
bts error - The value for the tempspace parameter, %s, is too long.

Part 5. Hierarchical Data Type

Chapter 19. Node DataBlade Module for Querying Hierarchical Data

This chapter provides information about what the capabilities of the Node DataBlade module, system requirements, and an introduction to the use of the Node DataBlade module functions.

The Node DataBlade module 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 DataBlade module to improve query performance for many recursive queries. Using the Node DataBlade module 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 DataBlade module introduces the node data type, which is an opaque type of variable length up to 256 characters. Operations involving ER replication are supported. However, deep copy and LIKE matching statements are not supported.

Node DataBlade Module Prerequisites

The Node DataBlade Module, Version 1.0 was released as an unsupported DataBlade module on IBM developerWorks® in 2001. Version 2.0 has the following enhancements:

- Support for Enterprise Replication (ER)
- New **depth()** function, which has the same functionality as the **length()** function
- Maximum node size increased from 64 to 256 bytes
- New **noderelease()** function
- Error number prefix changes from UNOD to UNDE
- Additional trace functions

Direct upgrades from Version 1.0 to Version 2.0 are not supported because unpredictable results can occur if you customized Version 1.0. Use caution if you customized Version 1.0 to allow for a node length greater than 256 bytes because data truncation might occur. You can use the character LENGTH() function to determine the maximum size of your node data

To upgrade from Node DataBlade Module, Version 1.0 to Version 2.0, follow these steps:

1. Unload the data.

2. Unregister the Node DataBlade module, Version 1.0 module with BladeManager.
3. Install Node DataBlade module, Version 2.0.
4. Register Node DataBlade module, Version 2.0 with BladeManager.
5. Reload the data.

The Node DataBlade Module requires IBM Informix Dynamic Server, Version, 11.10 or later. See the *IBM Informix DataBlade Module Installation and Registration Guide* for more information on registering DataBlade modules.

Troubleshooting the Node DataBlade Module

You might receive the following errors:

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 *IBM Informix Guide to SQL: Reference*.

Chapter 20. Node DataBlade Functions

The Node DataBlade provides a number of functions for robust operation.

Ancestors() Node DataBlade 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

Example 1:

```
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() Node DataBlade 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

Example 1:

```
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() Node DataBlade 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.

Example

Example 1:

```
EXECUTE FUNCTION DEPTH('1.22.3');
```

Returns: 3

Example 2:

```
EXECUTE FUNCTION DEPTH('6.5.4.3.2.1');
```

Returns: 6

Equal() Node DataBlade Function

The **Equal( )** function compares two variable-length opaque types This function implements the comparison operator, so you can use it in SQL statements using the function name or the corresponding symbol.

Returns: Boolean

Syntax

`Equal(`*node1*`,`*node2*`)`

node
The node against which you will test for equality.

node2
The node that you will compare to the first to test for equality.

Example

Example 1:

```
SELECT * FROM tablename WHERE Equal(nodecolumn, "1.4");
```

Example 2:

```
SELECT * FROM tablename WHERE nodecolumn = "1.4";
```

This example is the same as Example 1, except an equals sign is used.

GetMember() Node DataBlade 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

Example 1:

```
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() Node DataBlade 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

Example 1:

```
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)
```

Graft() Node DataBlade Function

The **Graft( )** function moves parts of the node tree. The Graft() function is useful for moving subsections of the tree and returns a new node value that is the result of grafting the third argument, under the second argument, from the first argument node down. No values are verified against any table data.

Returns: node

Syntax

Graft(*node1*, *node2*, *node3*)

node1
The parent of the node that you are grafting to another location.

node2
The new parent of the grafted node.

node3
The node to move from a child of node1 to a child of node2.

Example

Example 1:

```
EXECUTE FUNCTION Graft ("1.2.3", "1.4", "1.2.3.2");
(expression) 1.4.2
```

The node 1.2.3.2 is moved from under node 1.2.3 to under node 1.4. The moved node becomes 1.4.2. Existing nodes cannot be overwritten.

GreaterThan() Node DataBlade Function

The **GreaterThan( )** function compares two nodes to determine which is greater. This function implements the comparison operator and can be used in SQL statements either using the function name or the corresponding symbol.

Returns: Boolean

Syntax

GreaterThan(*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 node1.

Example

Example 1:

```
SELECT *
FROM tablename
WHERE GreaterThan(nodecolumn, "1.4");
```

Example 2:

```
SELECT *
FROM tablename
WHERE nodecolumn > "1.4";
```

This example is the same as Example 1, except a greater than sign is used in place of the function name.

GreaterThanOrEqual() Node DataBlade 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.

Example

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() Node DataBlade 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.

Example

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() Node DataBlade 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

IsAncestor*node1*, *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.

Example

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() Node DataBlade 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

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, 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() Node DataBlade 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

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, 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() Node DataBlade 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

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, 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 DataBlade Function

The **Length( )** 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
The node for which you want to determine depth, which is how many levels are in the node.

Example

Example 1:

```
execute function length('1.22.3');
(expression)  3
```

LessThan() Node DataBlade 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.

Example

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() Node DataBlade 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.

Example

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() Node DataBlade 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

Example 1:

```
EXECUTE FUNCTION NewLevel ('1.2.3');
(expression)  1.2.3.1
```

NodeRelease() Node DataBlade Function

The **NodeRelease( )** function reports the release and version information of the node DataBlade. This function takes no arguments.

Returns: string

Syntax

```
NodeRelease()
```

node

Example

NotEqual() Node DataBlade 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.

Example

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.

Part 6. Web Feature Service for Geospatial Data

Chapter 21. Web Feature Service DataBlade Module Administration

The Web Feature Service (WFS) DataBlade module lets you add an Open Geospatial Consortium (OGC) web feature service as a presentation layer for the Spatial and Geodetic DataBlade modules. See the *Release Notes* for the Spatial and Geodetic DataBlade modules for details on their support of WFS. An OGC web feature service allows requests for geographical features across the web using platform-independent calls.

The WFS DataBlade module includes support for inserting, updating, and deleting features using a CGI client program, wfsdriver, and a server-side function, WFSExplode().

A web feature service (WFS) handles requests for geographical features from a web server using platform-independent calls. The Web Feature Service (WFS) DataBlade Module is based on the transaction WFS specification from the Open Geospatial Consortium (OGC). See the *Release Notes* for the Spatial and Geodetic DataBlade modules for details on their support of WFS. You can use these DataBlade modules to let you support web-based geographical programs using data that you have stored in Dynamic Server. You can insert, update, and delete geographical features. The XML-based Geography Markup Language (GML) encodes the geographic features. The detailed specification is available at www.opengeospatial.org.

Requirements

To use this DataBlade module, you must meet these requirements:

- Install IBM Informix Dynamic Server, Version 11.10 or later
- Install and register either the Spatial or Geodetic DataBlade modules in the same database as the WFS DataBlade module

See the *Release Notes* for the Spatial and Geodetic DataBlade modules for details on their support of WFS.

This DataBlade module encodes geographic features in the Geography Markup Language (GML) 3.1.1 specification. GML 2.1.1 is also supported for compatibility. All features must be uniquely identified. The identifiers commonly take the form of `Feature.ObjectID`, where `Feature` is a feature class or table and `ObjectID` is a unique identifier (usually a primary key) for that class or table.

For detailed information about operating systems on which you can run the WFS DataBlade module, see the *Release Notes*.

The WFSDriver CGI Program

The WFSDriver CGI program processes all requests using either the HTTP methods GET or POST encoded as key-value-pairs (KVP) or XML. The program uses its corresponding wfs.cnf file to determine which Informix database to connect to, how to connect to it, and the user ID to use to connect to the database.

The WFSDriver CGI program determine whether it is passing KVP or XML data. KVP data goes through preliminary validation, while XML is passed directly to the wfsexplode UDR on the data server. The WFSDriver CGI program finally returns the results from the WFSExplode UDR and returns them to the web server.

Configuring the WFSDriver Program

Before your web server can run the CGI program, you must configure your web server to recognize the path in which the CGI program runs. For example, on an Apache web server with a root directory `/local0/IBMIHS` and a database name `mywfs`, the WFSSetup program creates a directory `/local0/IBMIHS/mywfs`, which contains the files wfs.cnf and wfsdriver.

You must edit the web server configuration file, httpd.conf, in `/local0/IBMIHS/conf` and add the following line so the web server can find the CGI program:

```
ScriptAlias /mywfs "/local0/IBMIHS/mywfs/"
```

Other web servers might use somewhat different configuration formats. See your web server documentation for configuration details.

Complete the following steps to configure your system:

1. Install and register the Web Feature Service DataBlade Module.
2. Run WFSSetup as described in "WFSSetup Program" on page 22-9.
3. Run WFSRegister on the tables on which you want to use the web feature service. See "WFSRegister UDR" on page 22-9 for details.
4. Edit your web server's configuration file and add permission to run the wfsdriver CGI program from its existing location.

WFS DataBlade Module Transactions

The transaction operation includes insert, update, and delete operations on web-accessible feature instances. After a transaction completes, the WFS DataBlade Module generates an XML response document that indicates the completion status of the transaction. A transaction operation can contain multiple insert, update, and delete elements. These elements are processed in the order in which they are contained in the transaction request.

The `TransactionResponse` element contains a `TransactionSummary` element, and the optional `TransactionResult` and `InsertResults` elements. The results of a transaction request are summarized in the `TransactionSummary` element in the `totalInserted`, `totalUpdated`, and `totalDeleted` elements. The optional `TransactionResult` element is required. The contents of the `TransactionResult` element indicates which actions of the transaction request failed to complete successfully. For details on transaction operations, see "WFS Transactions" on page 22-4.

Implementing Security in the WFS DataBlade Module

The web server handles secure access to the CGI program. The password to access the database is stored in the wfs.cnf file, which is in the same directory as the WFSDriver CGI program. The user ID should have permission to select, insert, update, and delete features. You can use the WFSpwcrypt program to generate encrypted passwords for the user IDs. See "WFSpwcrypt program" on page 22-9 for more information.

Chapter 22. Web Feature Service DataBlade Module Reference

DescribeFeatureType Element

A DescribeFeatureType request contains zero or more `TypeName` elements that encode the names of feature types that are to be described. This request is the same as issuing the following query in dbaccess:

```
INFO COLUMNS FOR TABLE tableName
```

If the content of the DescribeFeatureType element is empty, all of the feature types (that is, tables) that are registered to the WFS are returned. The following XML schema fragment defines the XML encoding of a DescribeFeatureType request:

```
<xsd:element name="DescribeFeatureType" type="wfs:DescribeFeatureTypeType"/>
<xsd:complexType name="DescribeFeatureTypeType">
   <xsd:complexContent>
      <xsd:extension base="sfs:BaseRequestType">
         <xsd:sequence>
            <xsd:element name="TypeName" type="xsd:QName"
                         minOccurs="0" maxOccurs="unbounded"/>
         </xsd:sequence>
         <xsd:attribute name="outputFormat"
                        type="xsd:string" use="optional"
                        default="text/xml; subtype=gml/3.1.1"/>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>
```

The following example shows a DescribeFeatureType request with its key-value pairs:

```
http://www.ibm.com/mydb/wfsdriver.cgi?SERVICE=WFS&VERSION=1.1.0&
REQUEST=DescribeFeatureType&TypeName=TreesA_1M
```

GetCapabilities Element

The web feature service (WFS) can describe its capabilities by returning service metadata in response to a GetCapabilities request. A GetCapabilities request uses key-value pair (KVP) encoded form over an HTTP GET request.

Table 22-1. Keys of GetCapabilities

Key	Mandatory or Optional	Definition and Example
service	Mandatory	Service type identifier. `service=WFS`
request	Mandatory	Operation name `request=GetCapabilities`
AcceptVersions	Optional. Returns the latest supported version if omitted.	Comma-separated prioritized sequence of one of more specification versions accepted by the client, with preferred versions listed first. `AcceptVersions=1.1.0,1.0,0`
updateSequence	Optional. Returns the most recent metadata document version if omitted or not supported by the web server.	Service metadata document version. The value is increased whenever any change is made in complete metadata document. `updateSequence=123`
AcceptFormats	Optional. Returns a service metadata document using MIME types `text/xml` if omitted or not supported by the web server.	A comma-separated sequence of zero or more response formats for the client. List the preferred formats first. `AcceptFormats=text/xml`

The following example shows a GetCapabilities request that is encoded using KVP:

```
http://hostname:port/wfsdriver.cgi?SERVICE=WFS&REQUEST=GetCapabilties&
ACCEPTVERSIONS=1.1.0,1.0.0&SECTIONS=Contents&UPDATESEQUENCE=XXX&
ACCEPTFORMATS=text/xml
```

The response document contains the following sections:

1. Service identification
2. Service provider
3. Operational metadata
4. FeatureType list
5. ServesGMLObjectType list
6. SupportsGMLObjectType list
7. Filter capabilities

GetFeature

The GetFeature operation lets you retrieve features from a WFS. The information that is retrieved can be features or a number that indicates how many features match your query. You can use the `MaxFeatures` element to limit the number of features that are returned.

The GetFeature operation contains one or more Query elements, each of which contains the description of the query. The results of all queries in a GetFeature request are concatenated into a result set. The typeName attribute in the schema indicates the name of one or more feature type instances or class instances to be queried. The value of this attribute is a list of valid feature types that are registered in the database. Specifying more than one typeName indicates that a join operation is being performed on the relational tables of the database.

The XML encoding of a GetFeature request is defined by the following XML schema fragment:

```
<xsd:element name="GetFeature" type="wfs:GetFeatureType"/>
<xsd:complexType name="GetFeatureType">
   <xsd:complexContent>
      <xsd:extensions base="wfs:BaseReqeustType">
         <xsd:sqeuence>
            <xsd:element ref="wfs:Query" maxOccursj="unbounded"/>
         </xsd:squence>
         <xsd:attribute name="resultType" type="wfs:ResultTypeType"
                        use="optional" default="results"/>
         <xsd:attribute name="outputFormat" type="xsd:string"
                        use="optional" default="text/xml; subtype=3.1.1"/>
         <xsd:attribute name="traverseXlinkDepth" type="xsd:string"
                        use="optional"/>
         <xsd:attribute name="traverseXlinkExpiry" type="xsd:positiveIngeger"
                        use="optional"/>
      </xsd:extension>
   </xsd:complexContent>
</xsd:complexType>

<xsd:simpleType name="ResultTypeType">
   <xsd:restriction base="xsd:string">
      <xsd:enumeration value="resuls"/>
      <xsd:enumeration value="hits"/>
   </xsd:restriction>
</xsd:simpleType>

<xsd:element name="Query type="wfs:QueryType"/>
<xsd:complexType name="QueryType">
   <xsd:sequence>
      <xsd:choice minOccurs="0" maxOccurs="unbounded">
         <xsd:element ref="wfs:PropertyName"/>
         <xsd:element ref="ogs:Function"/>
      </xsd:choice>
      <xsd:element ref="ogc:Filter" minOccurs="0" MaxOccurs="1"/>
      <xsd:element ref="ogc:SortBy" minOccurs="0" MaxOccurs="1"/>
   </xsd:squence>
   <xsd:attribute name="handle" type="xsd:string" use="optional"/>
   <xsd:attribute name="typeName" type="wfs:TypeNameListType" use="required"/>
   <xsd:attribute name="featureVersion" type="xsd:string" use="optional"/>
</xsd:complexType>
<xsd:simpleType name="Base_TypeNameListType">
   <xsd:list itemType="OName"/>
<.xsd:simpleType>
<xsd:simpleType name="TypeNameListType">
   <xsd:restriction base="wfs:Base_TypeNameListType">
      <xsd:pattern value="([\w:)?\w((=\w)?]{1,}"/>
   </xsd:restriction>
</xsd:simpleType>
```

The following query returns all properties of all instances of type InWaterA_1M:

```
http://www.ibm.com/wfsdriver.cgi&SERVICE=WFS&VERSION=1.1.0&
REQUEST=GetFeature&TypeName=InWaterA_1M
```

The query is passed to the WFSExplode UDR, which creates the following SQL query:

```
SELECT genxmlclob('InWaterA_1M',ROW(id,tileid,GeoASGML(geom)))
FROM InWaterA_1M;
```

WFS Transactions

If a transaction request includes a insert operation, the unique feature identifier is reported for each operation that was part of the transaction. The following XML schema fragment shows the XML coding of a WFS transaction response:

```
<xsd:element name="TransactionResponse" type="wfs:TransactionResponseType"/>
<xsd:complexType name="TransactionResponseType">
   <xsd:sequence>
      <xsd:element name="TransactionSummary" type="wfs:TransactionSummaryType"/>
      <xsd:element name="TransactionResults" type="wfs:TransactionResultsType"
                   minOccurs="0"/>
      <xsd:element name="InsertResults" type="wfs:InsertResultsType" minOccurs="0"/>
   </xds:sequence>
   <xsd:attribute name="version" type="xsd:string" use="required" fixed="1.1.0"/>
</xsd:complexType>
<xsd:complexType name="TransactionSummaryType">
   <xsd:sequence>
      <xsd:element name="totalInserted"
                   type="xsd:nonNegativeInteger" minOccurs="0"/>
      <xsd:element name="totalUpdated"
                   type="xsd:nonNegativeInteger" minOccurs="0"/>
      <xsd:element name="totalDeleted"
                   type="xsd:nonNegativeInteger" minOccurs="0"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType>
<xsd:complexType name="TransactionResultsType">
   <xsd:sequence>
      <xsd:element name="Action" type="wfs:ActionType" minOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="ActionType">
   <xsd:sequence>
      <xsd:element name="Message" type="xsd:string" minOccurs="0" maxOccurs="1"/>
   </xsd:sequence>
   <xsd:attribute name="locator" type="xsd:string" use="required"/>
   <xsd:attribute name="code" type="xsd:string" use="optional"/>
</xsd:complexType>
<xsd:complexType name="InsertResultsType">
   <xsd:sequence>
      <xsd:element name="Feature" type="wfs:InsertedFeatureType"
                   maxOccurs="unbounded"/>
   </xsd:sequence>
</xsd:complexType>
<xsd:complexType name="InsertedFeatureType">
   <xsd:sequence>
      <xsd:element ref="ogc:FeatureId" maxOccurs="unbounded"/>
   </xsd:sequence>
   <xsd:attribute name="handle" type="xsd:string" use="optional"/>
</xsd:complexType>
```

Insert Element

The `Insert` element creates new feature instances. By default, the initial state of a feature to be created is expressed using GML3, but the defined `inputFormat` attribute supports older versions of GML. In response to an insert operation, the WFS generates a list of identifiers assigned to the new feature instances. Feature identifiers are generated by the WFS or specified by the client using `gml:id` attribute values on inserted features and elements. The `idgen` attribute defined on

the Insert element can indicate a method of assigning feature identifiers to use, as shown in the following table.

Table 22-2. Actions Corresponding to idgen Values

idgen Value	Action
GenerateNew (default)	The WFS generates unique identifiers for all newly inserted feature instances.
UseExisting	In response to an insert operation, the web feature service uses the gml:id attribute values on inserted features and elements. If any IDs duplicate the ID of a feature or element already stored in the WFS, the WFS raises an exception.
ReplaceDuplicate	A WFS client can request that the WFS generate IDs to replace the input values of gml:id attributes of feature elements that duplicate the ID of a feature or element already stored in the WFS instead of raising an exception by setting the idgen attribute of the InsertElementType to the value ReplaceDuplicate.

After an insert operation, the WFS generates a list of identifiers that are assigned to the new feature instances. The following example shows an insert operation:

```
<wfs:Transaction
 version="1.1.0"
 service="WFS"
 handle="Transaction 01"
 xmlns="http://www.yourserver.com/mydbns"
 xmlns:wfs="http://www.opengis.net/wfs"
 xmsns:ogc="http://www.opengis.net/ogc"
 xmsns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemalocation="http://www.yourserver.com/mydbns

http://www.yourserver.com/wfs/wfs.cgi?request=DESCRIBEFEATURETYPE&
typename=ELEVP_1M

http://www.opengis.net/wfs ../wfs/1.1.0/WFS.xsd">

<wfs:Insert handle="statement 1">
<ElevP_1M>
   <id>167928</id>
   <f_code>CA</fcode>
   <acc>2</acc>
   <ela>1</ela>
   <ZV2>152</ZV2>
   <tileID>250</tileID>
   <end_id>111</end_id>
   <location>
    <gml:Point srsname="http://www.opengis.net/gms/srs/epsg.xml#63266405">
    <gml:pos>-98.5485 24.2633</gml:pos>
    </gml:Point>
   </location>
  </ElevP_1M>
</wfs:Insert>
  </wfs:Transaction>
```

The WFSExplode() function transforms the insert operation into the following INSERT statement:

```
INSERT INTO ElevP_1M
   (id,f_code,acc,ela,ZV2,tileID,end_id,location)
VALUES (167928,'CA',2,1,152, 250, 111,
   GeoFromGML('<gml:Point ...> ... </gml:Point>')
```

Update Element

The Update element describes one update operation to apply to a feature or set of features of a single feature type. Multiple update operations can be contained in a single transaction request. The Filter element can limit the scope of an update operation to a numbered set of features using spatial and non-spatial constraints. The following is an example of an update transaction that is filtered by a non-spatial constraint:

```
<?xml version="1.0" ?>
<wfs:Transaction
 version="1.1.0"
 service="WFS"
 handle="Transaction 01"
 xmlns="http://www.yourserver.com/mydbns"
 xmlns:wfs="http://www.opengis.net/wfs"
 xmsns:ogc="http://www.opengis.net/ogc"
 xmsns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemalocation="http://www.yourserver.com/mydbns
  http://www.yourserver.com/wfs/wfs.cgi?request=DESCRIBEFEATURETYPE&amp;
typename=BuiltUpA_1M
 http://www.opengis.net/wfs ../wfs/1.1.0/WFS.xsd">

<wfs:Update typename="BuiltUpA_1M>
 <wfs:Property>
  <wfs:Name>bndry</wfs:Name>
  <wfs:Value>
  <gml:Polygon gid="g5"
  srsname="http://www.opengis.net/gml/srs/epsg.xml#63266405">
  <gml:exterior>
   <gml:LinearRing>
   <gml:PosList>-89.8 44.3 -89.9 44.4 ... </gml:PosList>
   </gml:LinearRing>
  </gml:exterior>
  </gml:Polygon>
  </wfs:Value>
 </wfs:Property>
 <ogc:Filter>
  <ogc:GmlObjectId gml:id="BuiltUpA_1M.1725"/>
 </ogc:Filter>
</wfs:Update>
  </wfs:Transaction>
```

The WFSExplode() function transforms the request into the following UPDATE statement:

```
UPDATE BuiltUpA_1M
SET bndry=GeoFromGML('<:gml:Polygon ...> ... </gml:Polygon>)
WHERE id=1725;
```

If the Filter element does not identify any feature instances on which to operate, no result is returned and no exception is raised.

Delete Element

The Delete element is used to delete one or more feature instances. The scope of the delete is determined by using the Filter element similar to how the Update element is constrained. If the Filter element does not identify any feature instances on which to operate, no result is returned and no exception is raised. The Delete element is a special case within the transaction operation, because it is the

only element that can be specified by either the XML or KVP encoding methods. The first example is XML encoded delete operation; the second is a KVP encoded delete operation:

```
<wfs:Transaction
 version="1.1.0"
 service="WFS"
 handle="Transaction 01"
 xmlns="http://www.yourserver.com/mydbns"
 xmlns:wfs="http://www.opengis.net/wfs"
 xmsns:ogc="http://www.opengis.net/ogc"
 xmsns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemalocation="http://www.yourserver.com/mydbns

http://www.yourserver.com/wfsdriver.cgi?request=DESCRIBEFEATURETYPE&amp;
typename=BuiltUpA_1M
 http://www.opengis.net/wfs ../wfs/1.1.0/WFS.xsd">
 <wfs:Delete typeName="BuiltUpA_1M">
  <ogc:Filter>
   <ogc:GmlObjectID gml:id="BuiltUpA_1M.1013"/>
  </ogc:Filter>
 </wfs:Delete>
</wfs:Transaction>
```

KVP encoded delete operation:

```
http://www.yourserver.com/wfsdriver.cgi?
 SERVICE=WFS&
 VERSION=1.1.0&
 REQUEST=Transaction&
 OPERATION=Delete&
 FEATUREID=BuiltUpA_1M.1013
```

WFSExplode generates the same DELETE statement in both cases:

```
DELETE FROM BuiltUpA_1M WHERE id=1013
```

Native Element

The `Native` element allows access to vendor-specific capabilities of any particular web feature server or datastore. This element is defined by the following XML Schema fragment:

```
<xsd:element name="Native" type="wfs:NativeType"/>
 <xsd:complexType name="NativeType">
  <xsd: any />
  <xsd: attribute name="vendorId" type="xsd:string" use="required"/>
  <xsd: attribute name="safeToIgnore" types="xsd:Boolean" use="required"/>
 </xsd:complexType>
```

The `vendorId` attribute identifies the vendor that recognizes the command or operation enclosed by the `Native` element. The `safeToIgnore` attribute guides the actions of the WFS when the native operation is not recognized. The element can have the values `True` or `False`. The following example shows the `Native` element:

```
<Native vendorId="IBM Informix Dynamic Server WFS" safeToIgnore="True">
   execute function GeoParamSessionSet("WFSDisplayTemporal","true")
</Native>
```

WFS Transaction Response Document

The WFS generates an XML document that indicates the completion status of the transaction. If the transaction request includes an insert operation, the unique feature identifier is included for each operation that was part of the transaction. The following XML schema fragment defines the XML coding of the WFS transaction response document:

```
<xsd:element name="TransactionResponse" type="wfs:TransactionResponeType"/>
<xsd:complexType name="TransactionResponseType">
   <xsd:sequence>
      <xsd:element name="TransactionsSummary"
                   type="wfs:TransactionSummaryType"/>
      <xsd:element name="TransactionsResults"
                   type="TransactionResultsType" minOccurs="0"/>
      <xsd:element name="InsertResults"
                   type="InsertResultsType" minOccurs="0"/>
   </xsd:sequence>
   <xsd:attribute name="version"
                  type="xsd:string" use="required" fixed="1.1.0"/>
</xsd:complexType>
```

WFSConfig Program

Use this program to add a new path to the WFS web driver configuration file. The new path must include the following values:

- The database name
- The user ID
- The encrypted password
- The server name

The WFSConfig program has the following syntax:

```
wfsconfig -addmap -p path_name -f configpath_and_filename -d database -u userID
```

WFSExplode UDR

WFSExplode() is a Dynamic Server UDR that handles requests for displaying, creating, modifying, and deleting features that stored in the database. A request is passed to the WFSExplode() UDR after the web driver program, wfsdriver, validates the service and version of a request and determines if the request is GetCapbilities, DescribeFeatureType, GetFeature, Transaction, or another request in KVP format. The WFSExplode() UDR passes the returned data to the web server. The WFSExplode() UDR has two forms:

- The first form accepts an XML document from the WFSDriver program. It takes a CLOB or lvarchar type for the XML document in the following formats:

  ```
  WFSExplode(CLOB) returns ROW(lvarchar,CLOB)
  WFSExplode(lvarchar) returns ROW(lvarchar,CLOB)
  WFSExplode(Ivarchar,CLOB) returns ROW(lvarchar,CLOB)
  WFSExplode(lvarchar,lvarchar) returns ROW(lvarchar,CLOB)
  ```

  For example:

  ```
  execute function WFSExplode('GetCapabilties', NULL)

  execute function WFSExplode('DescribeFeatureType','TypeName=BuiltUpA_1M')

  execute function WFSExplode('GetFeature','TypeName=InWaterA_1M|
                   PropertyName=InWaterA_1M/wkbGeom/InWaterA_1M/tileId')

  execute function WFSExplode('Transaction',
                  'Operation=Delete|TypeName=InWaterA_1M|
                   Filter=(<:Filter><:Within><:PropertyName>InWaterA_1M/wkbGeom
                   <:/PropertyName><:gml:Envelope><:gml:lowerCorner>10 10
                   <:/gml:lowerCorner><:gml:upperCorner>20 20<:/gml:upperCorner>
                   <:/gml:Envelope><:/Within><:/Filter)')
  ```

- The second form takes 2 arguments in a key-value pair (KVP) format. The first argument will describe the transaction type (GetCapabilties, GetFeature,

DescribeFeatureType, Transaction), and the second argument is a list of additional parameters for the transaction that are separated by a vertical bar (|). For example:

```
WFSExplode('Transaction','Operation=Delete|FeatureId=BuiltUpA_1M')

execute function WFSExplode('GetFeature',
   'TypeName=InWaterA_1M|PropertyName=InWaterA_1M/wkbGeom/InWaterA_1M/tileId')
```

WFSpwcrypt program

The WFSpwcrypt program encrypts a password for the user ID that uses the web feature service. The WFS configuration file, wfs.cnf, includes the name of a database and the user ID with which the connection to the database is made. The WFS DataBlade Module automatically encrypts the password using its own encryption key. If, however, you want to use your own encryption key, you must use the webpwcrypt utility to create the encrypted password and update the web.cnf file manually. The webpwcrypt utility is located in the directory INFORMIXDIR/extend/ web.version/utils, where INFORMIXDIR refers to the main Informix directory and version refers to the current version of the Web DataBlade module installed on your computer.

```
wfspwcrypt database_name username key
```

WFSRegister UDR

This UDR makes sure that a table that contains features contains a primary key. All features that participate in a Web Feature Service must be able to be uniquely identified. Feature identifiers commonly take the form of `Feature.ObjectID`, where `Feature` is a feature class or table and `ObjectID` is a primary key for that class or table. WFSRegister() takes a single table name as its only argument. If the table does not have a primary key, an error is returned and the table cannot participate in the web feature service. WFSRegister() also verifies that there are no unsupported opaque types or IDS collection or row types in the table definition. Only Dynamic Server base types and the opaque types found in the Spatial or Geodetic DataBlade modules are supported.

Run the WFSRegister() UDR on a table before using it with the Web Feature Services DataBlade Module:

```
execute function WFSRegister(tableName)
```

WFSSetup Program

The WFSSetup program creates and configures the WFS configuration file, wfs.cnf. Determine the following values before you run the wfssetup program:

- INFORMIXDIR
- INFORMIXSERVER
- Web server directory
- Web driver type (The default is CGI.)
- Path name for URL WFS access
- Database name
- MI_WFSCONFIGDIR (For CGI the default is the web server CGI directory.)
- The user ID for connecting to database server
- The password that is associated with the user ID

The WFSSetup program copies the wfs.cnf and the web driver program, wfsdriver, to the path that you specified in MI_WFSCONGIDIR. The program prompts you to enter the password twice and will ask for a password key to use to encrypt the password.

To make changes to the values that you specified when you ran the WFSSetup program, run the WFSConfig program. See "WFSConfig Program" on page 22-8 for details.

Run the wfssetup program using the following syntax:

```
wfssetup [–s informix_server –w web_server –t driver_type –p path_name
         –d database  –u userID –c cnf_dir]
```

Part 7. Appendixes

Appendix. Accessibility

IBM strives to provide products with usable access for everyone, regardless of age or ability.

Accessibility features for IBM Informix Dynamic Server

Accessibility features help a user who has a physical disability, such as restricted mobility or limited vision, to use information technology products successfully.

Accessibility Features

The following list includes the major accessibility features in IBM Informix Dynamic Server. These features support:

- Keyboard-only operation.
- Interfaces that are commonly used by screen readers.
- The attachment of alternative input and output devices.

Tip: The IBM Informix Dynamic Server Information Center and its related publications are accessibility-enabled for the IBM Home Page Reader. You can operate all features using the keyboard instead of the mouse.

Keyboard Navigation

This product uses standard Microsoft® Windows® navigation keys.

Related Accessibility Information

IBM is committed to making our documentation accessible to persons with disabilities. Our publications are available in HTML format so that they can be accessed with assistive technology such as screen reader software. The syntax diagrams in our publications are available in dotted decimal format. For more information about the dotted decimal format, go to "Dotted Decimal Syntax Diagrams."

You can view the publications for IBM Informix Dynamic Server in Adobe Portable Document Format (PDF) using the Adobe Acrobat Reader.

IBM and Accessibility

See the *IBM Accessibility Center* at http://www.ibm.com/able for more information about the commitment that IBM has to accessibility.

Dotted Decimal Syntax Diagrams

The syntax diagrams in our publications are available in dotted decimal format, which is an accessible format that is available only if you are using a screen reader.

In dotted decimal format, each syntax element is written on a separate line. If two or more syntax elements are always present together (or always absent together), the elements can appear on the same line, because they can be considered as a single compound syntax element.

Each line starts with a dotted decimal number; for example, 3 or 3.1 or 3.1.1. To hear these numbers correctly, make sure that your screen reader is set to read punctuation. All syntax elements that have the same dotted decimal number (for example, all syntax elements that have the number 3.1) are mutually exclusive alternatives. If you hear the lines 3.1 USERID and 3.1 SYSTEMID, your syntax can include either USERID or SYSTEMID, but not both.

The dotted decimal numbering level denotes the level of nesting. For example, if a syntax element with dotted decimal number 3 is followed by a series of syntax elements with dotted decimal number 3.1, all the syntax elements numbered 3.1 are subordinate to the syntax element numbered 3.

Certain words and symbols are used next to the dotted decimal numbers to add information about the syntax elements. Occasionally, these words and symbols might occur at the beginning of the element itself. For ease of identification, if the word or symbol is a part of the syntax element, the word or symbol is preceded by the backslash (\) character. The * symbol can be used next to a dotted decimal number to indicate that the syntax element repeats. For example, syntax element *FILE with dotted decimal number 3 is read as 3 * FILE. Format 3* FILE indicates that syntax element FILE repeats. Format 3* * FILE indicates that syntax element * FILE repeats.

Characters such as commas, which are used to separate a string of syntax elements, are shown in the syntax just before the items they separate. These characters can appear on the same line as each item, or on a separate line with the same dotted decimal number as the relevant items. The line can also show another symbol that provides information about the syntax elements. For example, the lines 5.1*, 5.1 LASTRUN, and 5.1 DELETE mean that if you use more than one of the LASTRUN and DELETE syntax elements, the elements must be separated by a comma. If no separator is given, assume that you use a blank to separate each syntax element.

If a syntax element is preceded by the % symbol, that element is defined elsewhere. The string following the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 means that you should refer to a separate syntax fragment OP1.

The following words and symbols are used next to the dotted decimal numbers:

? Specifies an optional syntax element. A dotted decimal number followed by the ? symbol indicates that all the syntax elements with a corresponding dotted decimal number, and any subordinate syntax elements, are optional. If there is only one syntax element with a dotted decimal number, the ? symbol is displayed on the same line as the syntax element (for example, 5? NOTIFY). If there is more than one syntax element with a dotted decimal number, the ? symbol is displayed on a line by itself, followed by the syntax elements that are optional. For example, if you hear the lines 5 ?, 5 NOTIFY, and 5 UPDATE, you know that syntax elements NOTIFY and UPDATE are optional; that is, you can choose one or none of them. The ? symbol is equivalent to a bypass line in a railroad diagram.

! Specifies a default syntax element. A dotted decimal number followed by the ! symbol and a syntax element indicates that the syntax element is the default option for all syntax elements that share the same dotted decimal number. Only one of the syntax elements that share the same dotted decimal number can specify a ! symbol. For example, if you hear the lines

2? FILE, 2.1! (KEEP), and 2.1 (DELETE), you know that (KEEP) is the default option for the FILE keyword. In this example, if you include the FILE keyword but do not specify an option, default option KEEP is applied. A default option also applies to the next higher dotted decimal number. In this example, if the FILE keyword is omitted, default FILE(KEEP) is used. However, if you hear the lines 2? FILE, 2.1, 2.1.1! (KEEP), and 2.1.1 (DELETE), the default option KEEP only applies to the next higher dotted decimal number, 2.1 (which does not have an associated keyword), and does not apply to 2? FILE. Nothing is used if the keyword FILE is omitted.

* Specifies a syntax element that can be repeated zero or more times. A dotted decimal number followed by the * symbol indicates that this syntax element can be used zero or more times; that is, it is optional and can be repeated. For example, if you hear the line 5.1* data-area, you know that you can include more than one data area or you can include none. If you hear the lines 3*, 3 HOST, and 3 STATE, you know that you can include HOST, STATE, both together, or nothing.

Notes:

1. If a dotted decimal number has an asterisk (*) next to it and there is only one item with that dotted decimal number, you can repeat that same item more than once.
2. If a dotted decimal number has an asterisk next to it and several items have that dotted decimal number, you can use more than one item from the list, but you cannot use the items more than once each. In the previous example, you could write HOST STATE, but you could not write HOST HOST.
3. The * symbol is equivalent to a loop-back line in a railroad syntax diagram.

\+ Specifies a syntax element that must be included one or more times. A dotted decimal number followed by the + symbol indicates that this syntax element must be included one or more times. For example, if you hear the line 6.1+ data-area, you must include at least one data area. If you hear the lines 2+, 2 HOST, and 2 STATE, you know that you must include HOST, STATE, or both. As for the * symbol, you can only repeat a particular item if it is the only item with that dotted decimal number. The + symbol, like the * symbol, is equivalent to a loop-back line in a railroad syntax diagram.

Notices

This information was developed for products and services offered in the U.S.A.

IBM may not offer the products, services, or features discussed in this document in all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law:
INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
J46A/G4
555 Bailey Avenue
San Jose, CA 95141-1003
U.S.A.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this information and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement, or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

All IBM prices shown are IBM's suggested retail prices, are current and are subject to change without notice. Dealer prices may vary.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information contains sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. You may copy, modify, and distribute these sample programs in any form without payment to

IBM for the purposes of developing, using, marketing, or distributing application programs conforming to IBM's application programming interfaces.

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

> © (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. (enter the year or years). All rights reserved.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at http://www.ibm.com/legal/copytrade.shtml.

Adobe, the Adobe logo, and PostScript are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Intel, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Java and all Java-based trademarks are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, and Windows NT are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Other company, product, or service names may be trademarks or service marks of others.

Index

Special characters

A

B

C

D

E

F

G

H

I

L

M

N

O

P

Q

R

S

T

U

V

W

X

Printed in USA

SC23-9427-02

Spine information:

IBM Informix **Version 11.50**

IBM Informix Database Extensions User's Guide