Informix Product Family
Informix
Version 12.10

IBM Informix Database Extensions User's Guide

SC27-4512-02

Informix Product Family
Informix
Version 12.10

IBM Informix Database Extensions User's Guide

SC27-4512-02

Note

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

This edition replaces SC27-4512-01.

Contents

Introduction . . . . **ix**
In this introduction ix
About this publication ix
Types of users ix
What's New in Database Extensions for Informix database server, Version 12.10. ix
Example code conventions. x
Additional documentation xi
Compliance with industry standards xi
Syntax diagrams xi
How to read a command-line syntax diagram xii
Keywords and punctuation xiii
Identifiers and names. xiv
How to provide documentation feedback xiv

Part 1. Large object management

Chapter 1. About Large Object Locator **1-1**
Large object requirements 1-2

Chapter 2. Large Object Locator data types **2-1**
The lld_locator data type 2-1
The lld_lob data type 2-2

Chapter 3. Large Object Locator functions **3-1**
Interfaces 3-1
API library 3-1
ESQL/C library. 3-1
SQL interface 3-2
Working with large objects 3-2
The lld_close() function 3-3
The lld_copy() function 3-3
The lld_create() function 3-5
The lld_delete() function 3-7
The lld_open() function 3-8
The lld_read() function 3-10
The lld_seek() function 3-10
The lld_tell() function 3-12
The lld_write() function. 3-13
Client file support 3-14
The lld_create_client() function 3-14
The lld_delete_client() function 3-15
The lld_from_client() function 3-16
The lld_open_client() function 3-17
The lld_to_client() function 3-19
Error utility functions 3-20
The lld_error_raise() function 3-20
The lld_sqlstate() function 3-20
Smart large object functions 3-21
The LOCopy function 3-21
The LOToFile function 3-22
The LLD_LobType function 3-22

Chapter 4. Large Object Locator example code **4-1**
The SQL interface 4-1

The lld_lob type 4-1
The lld_locator type 4-3
The API interface 4-6
Create the lld_copy_subset function 4-6
The lld_copy_subset routine 4-9

Chapter 5. Large Object Locator error handling 5-1
Large Object Locator errors 5-1
Error handling exceptions 5-1
Error codes 5-2

Part 2. MQ Messaging

Chapter 6. About MQ messaging 6-1
Prepare to use MQ messaging 6-1
Install and configure WMQ 6-1
Prepare your database server for MQ messaging. 6-2
Sample code for setting up queue managers, queues, and channels 6-2
Sample code for setting up the server for use with WMQ. 6-3
Switch between server-based and client-based messaging 6-4
Verification 6-5
Insert data into a queue 6-5
Read an entry from a queue 6-5
Receive an entry from a queue. 6-6
Publish and subscribe to a queue 6-6

Chapter 7. MQ messaging tables 7-1
Schema mapping 7-1
General table behavior 7-1
Create and bind a table 7-1
Use INSERT and SELECT 7-2
Retrieve the queue element 7-2
Special considerations. 7-2
Table errors 7-3

Chapter 8. MQ messaging functions. 8-1
Service and policy tables. 8-1
The "informix".mqiservice table 8-2
The "informix".mqipubsub table 8-4
The "informix".mqipolicy table. 8-4
MQCreateVtiRead() function 8-8
MQCreateVtiReceive() function 8-10
MQCreateVtiWrite() function 8-12
MQHasMessage() function. 8-13
MQInquire() function 8-14
MQPublish() function 8-15
MQPublishClob() function 8-19
MQRead() function 8-23
MQReadClob() function 8-25
MQReceive() function 8-28
MQReceiveClob() function. 8-30
MQSend() function 8-32
MQSendClob() function. 8-35
MQSubscribe() function. 8-37
MQTrace() function 8-39
MQUnsubscribe() function. 8-41
MQVersion() function 8-42

Chapter 9. MQ messaging configuration parameters . . . 9-1
MQSERVER configuration parameter . . . 9-1
MQCHLLIB configuration parameter . . . 9-2
MQCHLTAB configuration parameter . . . 9-2

Chapter 10. MQ messaging error handling . . . 10-1

Chapter 11. Sample MQ messaging code . . . 11-1

Part 3. Binary data types

Chapter 12. Binary data types overview . . . 12-1

Chapter 13. Store and index binary data . . . 13-1
Binary data types . . . 13-1
  The binaryvar data type . . . 13-1
  The binary18 data type . . . 13-1
  ASCII representation of binary data types . . . 13-1
  Binary data type examples . . . 13-1
Insert binary data . . . 13-2
Index binary data . . . 13-3

Chapter 14. Binary data type functions . . . 14-1
Bitwise operation functions . . . 14-1
  The bit_and() function . . . 14-1
  The bit_complement() function . . . 14-1
  The bit_or() function . . . 14-2
  The bit_xor() function . . . 14-2
Support functions for binary data types . . . 14-3
  The bdtrelease() function . . . 14-3
  The bdttrace() function . . . 14-4
  The LENGTH() function . . . 14-4
  The OCTET_LENGTH() function . . . 14-4

Part 4. Basic Text Search

Chapter 15. Preparing for basic text searching . . . 15-1
Basic text search requirements and restrictions . . . 15-1
Creating a default sbspace . . . 15-2
Creating a space for the bts index . . . 15-3
Creating a space for temporary data . . . 15-4
Creating a bts index . . . 15-4
  bts access method syntax . . . 15-5

Chapter 16. Basic text search queries . . . 16-1
Basic Text Search query syntax . . . 16-1
Basic Text Search query terms . . . 16-2
Basic text search index fields . . . 16-3
Basic Text Search query term modifiers . . . 16-5
  Wildcard searches . . . 16-5
  Fuzzy searches . . . 16-6
  Proximity searches . . . 16-7
  Range searches . . . 16-8
  Boost a term . . . 16-8
Boolean operators . . . 16-9
  AND operator . . . 16-9
  OR operator . . . 16-10
  NOT operator . . . 16-10

Chapter 17. Basic text search JSON index parameters . . . 17-1
JSON index parameters syntax . . . 17-2
all_json_names index parameter . . . 17-4
ignore_json_format_errors index parameter . . . 17-5
include_contents index parameter . . . 17-5
json_array_processing index parameter . . . 17-6
json_names index parameter . . . 17-8
json_path_processing index parameter . . . 17-10
only_json_values index parameter . . . 17-11

Chapter 18. Basic Text Search XML index parameters. . . . 18-1
XML index parameters syntax . . . 18-1
The xmltags index parameter . . . 18-2
Example: Index specific XML tags . . . 18-4
The all_xmltags index parameter . . . 18-4
Example: Index all XML tags . . . 18-5
The all_xmlattrs index parameter . . . 18-5
Examples: Index XML attributes . . . 18-6
The xmlpath_processing index parameter . . . 18-7
Full paths and relative paths in path processing . . . 18-7
Example: Index XML paths . . . 18-8
The include_contents index parameter . . . 18-9
Example: Index XML tag values and XML tag names . . . 18-9
The strip_xmltags index parameter . . . 18-9
Example: Index XML tag values in a separate field . . . 18-10
The include_namespaces index parameter . . . 18-10
Example: Index namespaces in XML data . . . 18-11
The include_subtag_text index parameter . . . 18-12
Example: Index subtags in XML data . . . 18-12

Chapter 19. Basic text search analyzers . . . 19-1
analyzer index parameter . . . 19-1
Analyzer support for query and index options . . . 19-3
Alnum analyzer . . . 19-4
CJK analyzer . . . 19-4
eSoundex analyzer . . . 19-5
Keyword analyzer . . . 19-6
Simple analyzer . . . 19-7
Soundex analyzer . . . 19-7
Snowball analyzer . . . 19-8
Standard analyzer . . . 19-10
Stopword analyzer . . . 19-11
User-defined analyzer . . . 19-12
Whitespace analyzer . . . 19-13

Chapter 20. Basic text search functions. . . . 20-1
bts_index_compact() function . . . 20-1
bts_index_fields() function . . . 20-1
bts_release() function . . . 20-4
bts_tracefile() function . . . 20-5
bts_tracelevel() function . . . 20-5

Chapter 21. Basic text search performance . . . 21-1
Disk space for the bts index . . . 21-1
Adding BTS virtual processors to run multiple queries simultaneously . . . 21-2
Tune configuration parameters for basic text searching . . . 21-2

Chapter 22. Basic text search error codes 22-1

Part 5. Hierarchical data type

Chapter 23. The node data type for querying hierarchical data 23-1
Troubleshooting the node data type. 23-1

Chapter 24. Node data type functions. 24-1
Ancestors() function 24-1
Compare() function 24-1
Depth() function 24-2
Equal() function 24-2
GetMember() function 24-3
GetParent() function 24-3
Graft() function 24-4
GreaterThan() function 24-4
GreaterThanOrEqual() function 24-5
Increment() function 24-5
IsAncestor() function 24-6
IsChild() function. 24-7
IsDescendant() function. 24-8
IsParent() function 24-8
Length() Node function. 24-9
LessThan() function 24-9
LessThanOrEqual() function 24-10
NewLevel() function 24-10
NodeRelease() function 24-11
NotEqual() function. 24-11

Part 6. Informix web feature service for Geospatial Data

Chapter 25. Informix web feature service administration 25-1
The WFSDriver CGI program. 25-1
WFSVP virtual processor class 25-1
Configuring the WFSDriver program 25-2
WFS transactions 25-2
Implement security in WFS 25-2

Chapter 26. WFS reference 26-1
DescribeFeatureType element 26-1
GetCapabilities element. 26-1
GetFeature operation 26-2
WFS transactions 26-3
  Insert element 26-4
  Update element 26-5
  Delete element. 26-6
  Native element 26-7
  WFS transaction response document 26-7
WFSConfig program. 26-8
WFSExplode UDR 26-8
WFSpwcrypt program 26-9
WFSRegister UDR 26-9
WFSSetup program 26-9

Part 7. SQL Packages Extension

Chapter 27. DBMS_ALERT package 27-1
REGISTER procedure 27-1

REMOVE procedure . . . 27-1
REMOVEALL procedure . . . 27-2
SET_DEFAULTS . . . 27-2
SIGNAL procedure . . . 27-2
WAITANY procedure . . . 27-3
WAITONE procedure . . . 27-3

Chapter 28. DBMS_LOB package . . . 28-1
APPEND procedures . . . 28-2
COMPARE function . . . 28-2
COPY procedures. . . . 28-3
ERASE procedures . . . 28-4
GETLENGTH function . . . 28-4
INSTR function . . . 28-5
READ procedures . . . 28-5
SUBSTR function . . . 28-6
TRIM procedures . . . 28-6
WRITE procedures . . . 28-7

Chapter 29. DBMS_OUTPUT package . . . 29-1
DISABLE procedure . . . 29-1
ENABLE procedure . . . 29-2
GET_LINE procedure . . . 29-2
GET_LINES procedure . . . 29-2
NEW_LINE procedure . . . 29-3
PUT procedure . . . 29-3
PUT_LINE procedure . . . 29-3

Chapter 30. DBMS_RANDOM package . . . 30-1
INITIALIZE procedure . . . 30-1
SEED procedure . . . 30-1
RANDOM function . . . 30-2
TERMINATE procedure . . . 30-2

Chapter 31. UTL_FILE package . . . 31-1
FCLOSE procedure . . . 31-2
FCLOSE_ALL procedure . . . 31-2
FFLUSH procedure . . . 31-2
FOPEN function . . . 31-2
GET_LINE procedure . . . 31-3
NEW_LINE procedure . . . 31-4
PUT procedure . . . 31-4

Part 8. Appendixes

Appendix. Accessibility . . . A-1
Accessibility features for IBM Informix products . . . A-1
  Accessibility features . . . A-1
  Keyboard navigation . . . A-1
  Related accessibility information . . . A-1
  IBM and accessibility. . . . A-1
Dotted decimal syntax diagrams . . . A-1

Notices . . . B-1
Privacy policy considerations . . . B-3
Trademarks . . . B-3

Index . . . X-1

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 database extensions that come with IBM® Informix®:

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

Types of users

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

What's New in Database Extensions for Informix database server, Version 12.10

This publication includes information about new features and changes in existing functionality.

The following changes and enhancements are relevant to this publication. For a complete list of what's new in this release, go to http://pic.dhe.ibm.com/infocenter/informix/v121/topic/com.ibm.po.doc/new_features_ce.htm.

Table 1. What's New in IBM Informix Database Extensions User's Guide for Version 12.10.xC4

Overview	Reference
Basic text searching support for JSON and BSON data You can now create a basic text search index on columns that have JSON or BSON data types. You can create the basic text search index on JSON or BSON data types through SQL with the CREATE INDEX statement or on BSON data types through the Informix extension to MongoDB with the **createTextIndex** command. You can control how JSON and BSON columns are indexed by including JSON index parameters when you create the basic text search index. You can run a basic text query on JSON or BSON data with the **bts_contains()** search predicate in SQL queries or the **$ifxtext** query operator in JSON queries.	Chapter 17, "Basic text search JSON index parameters," on page 17-1

Table 2. What's New in IBM Informix Database Extensions User's Guide for Version 12.10.xC2

Overview	Reference
Enhanced basic text searching You have several new options to customize basic text searching when you create a **bts** index. You can increase the maximum number of tokens to index in a document with the **field_max_token** index parameter, instead of being limited to the previous maximum of 10 000 tokens. You can build the **bts** index faster in RAM than in a temporary sbspace by including the **xact_ramdirectory="yes"** index parameter. You can limit the amount of memory that is available for basic text search operations with the **xact_memory** index parameter. You can index words that contain numbers and other characters by specifying the Alnum analyzer.	"xact_ramdirectory index parameter" on page 15-19 "xact_memory index parameter" on page 15-18 "field_token_max index parameter" on page 15-12 "Alnum analyzer" on page 19-4

Table 3. What's New in IBM Informix Database Extensions User's Guide for Version 12.10.xC1

Overview	Reference
SPL routines for application compatibility The SQL packages extension provides SPL (Stored Procedure Language) routines that you can use in an application that is compatible with other database servers. For example, the packages include large object handling, alert and message management, and random number generation.	Part 7, "SQL Packages Extension"

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 that is 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 this release of IBM Informix products is available in various formats.

You can access Informix technical information such as information centers, technotes, white papers, and IBM Redbooks® publications online at http://www.ibm.com/software/data/sw-library/.

Compliance with industry standards

IBM Informix products are compliant with various standards.

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

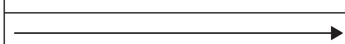
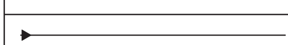
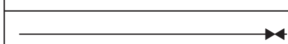
Component represented in PDF	Component represented in HTML	Meaning
	`>>----------------------`	Statement begins.
	`----------------------->`	Statement continues on next line.
	`>-----------------------`	Statement continues from previous line.
	`----------------------><`	Statement ends.
——— SELECT ———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

Table 4. Syntax Diagram Components (continued)

<table>
<tr><th>Component represented in PDF</th><th>Component represented in HTML</th><th>Meaning</th></tr>
<tr><td>LOCAL</td><td><pre>--+------------------+---
  '------LOCAL------'</pre></td><td>Optional item.</td></tr>
<tr><td>ALL
DISTINCT
UNIQUE</td><td><pre>---+-----ALL-------+---
   +--DISTINCT-----+
   '---UNIQUE------'</pre></td><td>Required item with choice. Only one item must be present.</td></tr>
<tr><td>FOR UPDATE
FOR READ ONLY</td><td><pre>---+------------------+---
    +--FOR UPDATE-----+
    '--FOR READ ONLY--'</pre></td><td>Optional items with choice are shown below the main line, one of which you might specify.</td></tr>
<tr><td>NEXT
PRIOR
PREVIOUS</td><td><pre>   .---NEXT---------.
----+----------------+---
    +---PRIOR--------+
    '---PREVIOUS-----'</pre></td><td>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 is used by default.</td></tr>
<tr><td>,
index_name
table_name</td><td><pre> .-------,-----------.
 V |
---+-----------------+---
    +---index_name---+
    '---table_name---'</pre></td><td>Optional items. Several items are allowed; a comma must precede each repetition.</td></tr>
<tr><td>Table Reference</td><td><pre>>>-| Table Reference |-><</pre></td><td>Reference to a syntax segment.</td></tr>
<tr><td>Table Reference
view
table
synonym</td><td><pre>Table Reference
|--+-----view--------+--|
   +------table------+
   '----synonym------'</pre></td><td>Syntax segment.</td></tr>
</table>

How to read a command-line syntax diagram

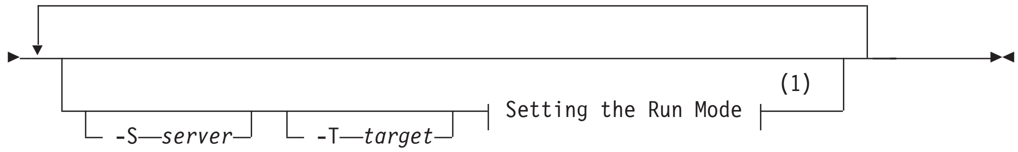
Command-line syntax diagrams use similar elements to those of other syntax diagrams.

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

Creating a no-conversion job

```
>>--onpladm create job--job--+------------------+-- -n-- -d--device-- -D--database------->
                             '-- -p--project----'

>-- -t--table------------------------------------------------------------------------------>
```

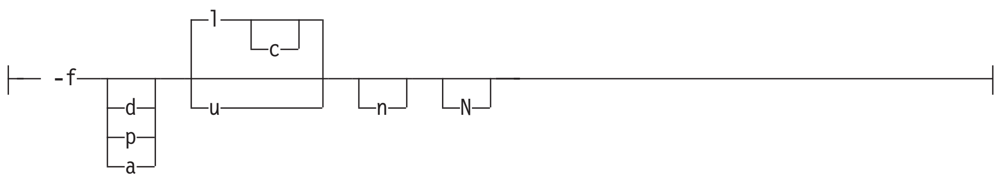


Notes:

1 See page Z-1

This diagram has a segment that is 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 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 upper 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. Include **onpladm create job** and then the name of the job.
2. Optionally, include **-p** and then the name of the project.
3. Include 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 include 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 include **-f**, optionally include **d**, **p**, or **a**, and then optionally include **l** or **u**.
5. Follow the diagram to the terminator.

Keywords and punctuation

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

A keyword in a syntax diagram 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 other syntax diagrams. A variable in a syntax diagram, an example, or text, is shown in *lowercase italic*.

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

```
►►──SELECT──column_name──FROM──table_name───────────────────────────────────►◄
```

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 email to docinf@us.ibm.com.
- In the Informix information center, which is available online at http://www.ibm.com/software/data/sw-library/, open the topic that you want to comment on. Click the feedback link at the bottom of the page, complete the form, and submit your feedback.
- Add comments to topics directly in the information center and read comments that were added by other users. Share information about the product documentation, participate in discussions with other users, rate topics, and more!

Feedback from all methods is monitored by the team that maintains the user documentation. The feedback methods are reserved for reporting errors and omissions in the documentation. For immediate help with a technical problem, contact IBM Technical Support at http://www.ibm.com/planetwide/.

We appreciate your suggestions.

Part 1. Large object management

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

IBM Informix 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 path name to it or identify it as a binary or character smart large object stored in the database. Smart large objects are a category of large objects that include CLOB and BLOB data types, which store text and images. Smart large objects are stored and retrieved in pieces, and have database properties such as crash recovery and transaction rollback.

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

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

Chapter 1. About Large Object Locator

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

The Large Object Locator data types are lld_locator and lld_lob.

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

You can also pass an lld_locator row to various Large Object Locator functions. For example, to create, delete, or copy a large object, and to open a large object for reading or writing, you pass an lld_locator row to the appropriate Large Object Locator function. See "The lld_locator data type" on page 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 "The lld_lob data type" on page 2-2 for a complete description of this data type.

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

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

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

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

Utility functions
: Raising errors and converting errors to their SQL state equivalents.

Smart large object functions
: Copying smart large objects to files and to other smart large objects

There are three interfaces to the Large Object Locator functions:
- An API library
- An ESQL/C library
- An SQL interface

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

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

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

Large object requirements

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

Database server requirements

The Informix database server has the following requirements:
- Non-logged databases are not supported.
- ANSI databases are not supported.
- The Scheduler must be running.

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

Transaction rollback

Because large objects, other than smart large objects, are stored outside the database, any changes to them take place outside the server's control and cannot be rolled back if a transaction is aborted. For example, when you execute **lld_create()**, it calls an operating system routine to create the large object itself. If you roll back the transaction containing the call to **lld_create()**, the server has no way of deleting the object that you have just created.

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

Concurrency control

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

Chapter 2. Large Object Locator data types

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

The lld_locator data type

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

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

lo_protocol
Identifies the kind of large object.

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

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

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

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

The *lo_pointer* field uses the lld_lob data type, which is defined by Large Object Locator. This data type allows you to point to a smart large object and specify whether it is of type BLOB or type CLOB. For more information, see "The lld_lob data type" on page 2-2.

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

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

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

Table 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 not case-sensitive. It is shown in uppercase letters for display purposes only.

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

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

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

You can pass an lld_locator type to these Large Object Locator functions, described in Chapter 3, "Large Object Locator functions," on page 3-1:
- "The lld_copy() function" on page 3-3
- "The lld_create() function" on page 3-5
- "The lld_delete() function" on page 3-7
- "The lld_open() function" on page 3-8
- "The lld_from_client() function" on page 3-16
- "The lld_to_client() function" on page 3-19

The lld_lob data type

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

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

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

It is defined for ESQL/C as follows:

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

lo A pointer to the location of the smart large object.

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

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

See "The lld_lob type" on page 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:

- "The LOCopy function" on page 3-21
- "The LOToFile function" on page 3-22
- "The LLD_LobType function" on page 3-22

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

Chapter 3. Large Object Locator functions

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

Interfaces

Large Object Locator functions are available through three interfaces:

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

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

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

API library

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

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

See "The API interface" on page 4-6 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 about using the ESQL/C interface to execute Large Object Locator functions.

SQL interface

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

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

See the following three-volume set for further information about the 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()
   Returns the current position within the large object.

   lld_seek()
   Changes the current position within the object.

   lld_read()
   Reads from large object.

   lld_write()
   Writes to the large object.

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

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

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

The lld_close() function

This function closes the specified large object.

Syntax

API

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

ESQL/C

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

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

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

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

Usage

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

Return codes

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

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

Context

"The lld_open() function" on page 3-8

The lld_copy() function

This function copies the specified large object.

Syntax

API

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

ESQL/C

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

SQL

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

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

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

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

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

Usage

This function copies an existing large object.

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

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

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

If you are copying to any type of large object other than a smart large object:
- specify `NULL` for the *lo_pointer* field.
- point to the location of the new object in the *lo_location* field.

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

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

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

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

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

Return codes

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

On failure, this function returns `NULL`.

Context

"The lld_from_client() function" on page 3-16

"The lld_to_client() function" on page 3-19

The lld_create() function

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

Syntax

API

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

ESQL/C

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

SQL

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

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

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

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

Usage

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

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

For any type of large object other than a smart large object:
- specify `NULL` for the *lo_pointer* field.
- point to the location of the new object in the *lo_location* field.

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

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

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

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

See “Large object requirements” on page 1-2 for more information.

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

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

Return codes

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

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

On failure, this function returns `NULL`.

Context

"The lld_delete() function"

"The lld_open() function" on page 3-8

The lld_delete() function

This function deletes the specified large object.

Syntax

API

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

ESQL/C

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

SQL

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

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

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

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

Usage

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

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

Return codes

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

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

The lld_open() function

This function opens the specified large object.

Syntax

API

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

ESQL/C

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

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

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

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

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

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

LLD_RDWR
Opens the large object for both reading and writing.

LLD_TRUNC
Clears the contents of the large object after opening.

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

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

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

Usage

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

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

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

See “Large object requirements” on page 1-2 for more information about transaction rollback and concurrency control.

Return codes

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

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

On failure, this function returns `NULL`.

Context

“The lld_close() function” on page 3-3

“The lld_create() function” on page 3-5

“The lld_read() function” on page 3-10

“The lld_seek() function” on page 3-10

“The lld_tell() function” on page 3-12

“The lld_write() function” on page 3-13

The lld_read() function

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

Syntax

API

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

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

ESQL/C

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

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

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

bytes The number of bytes to read.

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

Usage

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

Return codes

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

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

Context

"The lld_open() function" on page 3-8

"The lld_seek() function"

"The lld_tell() function" on page 3-12

The lld_seek() function

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

Syntax

API

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

ESQL/C

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

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

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

offset A pointer to the offset. It describes where to seek in the object. Its value depends on the value of the *whence* parameter.
- If *whence* is LLD_SEEK_SET, the offset is measured relative to the beginning of the object.
- If *whence* is LLD_SEEK_CUR, the offset is relative to the current position in the object.
- If *whence* is LLD_SEEK_END, the offset is relative to the end of the file.

whence Determines how the offset is interpreted.

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

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

Usage

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

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

Return codes

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

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

Context

"The lld_open() function" on page 3-8

"The lld_read() function" on page 3-10

"The lld_tell() function"

"The lld_write() function" on page 3-13

The lld_tell() function

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

Syntax

API

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

ESQL/C

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

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

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

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

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

Usage

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

Return codes

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

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

Context

"The lld_open() function" on page 3-8

"The lld_read() function" on page 3-10

"The lld_seek() function" on page 3-10

"The lld_write() function"

The lld_write() function

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

Syntax

API

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

ESQL/C

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

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

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

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

bytes The number of bytes to write.

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

Usage

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

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

Return codes

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

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

Context

"The lld_open() function" on page 3-8

"The lld_seek() function" on page 3-10

"The lld_tell() function" on page 3-12

Client file support

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

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

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

The lld_create_client() function

This function creates a new client file.

Syntax

API

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

ESQL/C

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

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

path A pointer to the path name of the client file.

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

Usage

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

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

See "Large object requirements" on page 1-2 for more information.

Return codes

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

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

Context

"The lld_delete_client() function"

The lld_delete_client() function

This function deletes the specified client file.

Syntax

API

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

ESQL/C

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

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

path A pointer to the path name of the client file.

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

Usage

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

Return codes

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

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

Context

"The lld_create_client() function" on page 3-14

The lld_from_client() function

This function copies a client file to a large object.

Syntax

API

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

ESQL/C

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

SQL

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

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

src A pointer to the source path name.

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

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

Usage

This function copies an existing large object.

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

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

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

If you are copying to any type of large object other than a smart large object:
- specify `NULL` for the *lo_pointer* field.

- point to the location of the new object in the *lo_location* field.

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

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

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

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

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

Return codes

On success, returns a pointer to an lld_locator row that specifies the location of the copy of the large object. If the destination object already exists, **lld_from_client()** returns a pointer to the unaltered lld_locator row that you created and passed in the *dest* parameter. If the destination object does not already exist, **lld_from_client()** returns an lld_locator row that points to the new object it creates.

On failure, this function returns NULL.

Context

"The lld_create_client() function" on page 3-14

"The lld_open_client() function"

The lld_open_client() function

This function opens a client file.

Syntax

API

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

ESQL/C

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

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

path A pointer to the path name of the client file.

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

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

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

LLD_RDWR
Opens the client file for both reading and writing.

LLD_TRUNC
Clears the contents of the client file after opening.

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

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

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

Usage

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

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

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

See "Large object requirements" on page 1-2 for more information about transaction rollback and concurrency control.

Return codes

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

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

On failure, this function returns `NULL`.

Context

"The lld_close() function" on page 3-3

"The lld_read() function" on page 3-10

"The lld_seek() function" on page 3-10

"The lld_tell() function" on page 3-12

"The lld_write() function" on page 3-13

"The lld_create_client() function" on page 3-14

The lld_to_client() function

This function copies a large object to a client file.

Syntax

API

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

ESQL/C

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

SQL

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

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

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

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

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

Usage

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

Return codes

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

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

Context

"The lld_open_client() function" on page 3-17

Error utility functions

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

The lld_error_raise() function

This function generates an exception for the specified error.

Syntax

API

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

error An error code that you specify.

Usage

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

Return codes

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

On failure, it also returns `MI_ERROR`.

The lld_sqlstate() function

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

Syntax

API

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

ESQL/C

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

error An error code.

Return codes

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

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

Smart large object functions

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

The LOCopy function

This function creates a copy of a smart large object.

Syntax

SQL

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

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

lob A pointer to the smart large object to copy.

table_name
A table name. This parameter is optional.

column_name
A column name. This parameter is optional.

Usage

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

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

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

See the description of the **LOCopy** function in the *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*.

The LOToFile function

Copies a smart large object to a file.

Syntax

SQL

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

lob A pointer to the smart large object.

pathname
A directory path and name of the file to create.

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

Usage

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

See the description of the **LOToFile** function in the *IBM Informix Guide to SQL: Syntax* for complete information about this function.

Return codes

This function returns the value of the new file name.

Context

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

The LLD_LobType function

Returns the type of data in an lld_lob column.

Syntax

SQL

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

lob A pointer to the smart large object

Usage

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

Return codes

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

Chapter 4. Large Object Locator example code

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

The SQL interface

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

The lld_lob type

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

Implicit lld_lob casts

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

```
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:00608460a6b7c8d900000000200000003000000020000001800000000000001000000608
     460736c6f000010029a2a6c920700000000000006c000af0cdd900000080006082500af0c9d
     e

key  2
slo  clob:00608460a6b7c8d900000000200000003000000030000001900000000000001000000608
     460736c6f000010029a2a6c930d000000000000006c000af0cdd900000016000000010af0c9d
     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.

Explicit lld_lob casts

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

This example uses the **slobs** table created in Figure 4-1 on page 4-1.

```
--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.

The LLD_LobType function

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

The **slobs** table in this example is the same one created in Figure 4-1 on page 4-1. 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. The 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.

The lld_locator type

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

lo_protocol
: Identifies the kind of large object.

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

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

The examples in this section show how to:

Insert an lld_locator row into a table

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

```
--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. Insert 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`.

Create a smart large object

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

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

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

```
--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

Copy a client file to a large object

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

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

```
-- 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
          0000090000001a000000000000100000000000000ad3c3dc000000000b06eec8000
          00000005c4e6000607fdc0000000000000000000000000',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. Copy 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.

Copy a large object to a large object

The following example uses the **lobs** table created in Figure 4-4 on page 4-3.

The **lld_copy** function in the example 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
          0000090000001a00000000000010000000000000ad3c3dc000000000b06eec8000
          00000005c4e6000607fdc000000000000000000000000',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. Copy 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.

Copy large object data to a client file

The following example uses the **lobs** table created in Figure 4-4 on page 4-3. The **lld_toclient** function in “Copy large object data to a client file” 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. Copy large object data to a client file

Create and delete a server file

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

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

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

```
-- 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. Create and delete 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.

The API interface

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

Create the lld_copy_subset function

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

```
/* LLD SAPI interface example */

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

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

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

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

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

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

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

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

```
    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

The **lld_copy_subset** function defines four parameters:

- A source large object (lld_locator type)
- A destination large object (lld_locator type)
- The byte offset to begin copying
- The number of bytes to copy

It returns an lld_locator, identifying the object being appended.

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

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

lld_open
: OpenS the source object

lld_seek
: Seeks to the specified byte offset in the object

lld_read
: Reads the specified number of bytes from the object

lld_close
: Closes the object

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

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

The **mi_close** function closes the database connection.

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

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

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

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

The lld_copy_subset routine

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

```
-- 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. 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

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

There are two methods by which Large Object Locator returns errors to you:
- Through the error argument of a Large Object Locator function
- Through an exception

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

Large Object Locator errors

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

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

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

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

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

Error handling exceptions

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

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

Error codes

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

Error code	SQL state	Description
LLD_E_INTERNAL	ULLD0	Internal Large Object Locator error. If you receive this error, call 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

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

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

Chapter 6. About MQ messaging

You can use either functions or tables to communicate between a database server application and an IBM WebSphere MQ queue.

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

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

Related reference:

Chapter 8, "MQ messaging functions," on page 8-1

Chapter 7, "MQ messaging tables," on page 7-1

Chapter 9, "MQ messaging configuration parameters," on page 9-1

Prepare to use MQ messaging

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

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

To use MQ messaging, you perform these tasks:

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

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

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

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

Install and configure WMQ

You must install and configure IBM WebSphere MQ before using MQ messaging.

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

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

As necessary, you need to complete the following WMQ queue configuration:
- Create a queue manager.
- Create a queue.
- Create a subscriber queue.

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

Prepare your database server for MQ messaging

You must prepare your Informix database for MQ messaging.

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

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

The Informix database server has the following additional requirements:
- Non-logged databases are not supported.
- ANSI databases are not supported.
- The Scheduler must be running.

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

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

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

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

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

1. Create queue manager `lqm1`, using`-q` to specify the default queue manager:
   ```
   crtmqm -q lqm1
   ```
2. Start the queue manager:
   ```
   strmqm lqm1
   ```

3. Start the publish/subscribe service:

```
strmqbrk -m lqm1
```

4. Stop the queue manager:

```
endmqm -w lqm1
```

5. Delete the queue manager:

```
dltmqm lqm1
```

6. Start the TCP listener on port 1414 for queue manager lqm1:

```
runmqlsr -t tcp -m lqm1 -p 1414 &
```

7. Run the following commands in runmqsc lqm1:

```
DEFINE CHANNEL(QM1CH) CHLTYPE(SVRCONN) TRPTYPE(TCP) +
DESCR('Server connection to WMQ client') REPLACE

DEFINE CHANNEL(QM1CH) CHLTYPE(CLNTCONN) TRPTYPE(TCP) +
CONNAME('hostname(1414)') +
DESCR('WebSphere MQ client connection to server 1') +
QMNAME('lqm1') REPLACE
```

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

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

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

Sample code for setting up the server for use with WMQ

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

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

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

```
-- Service for most operations

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

-- Service for publishing

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

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

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

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

-- publisher information
```

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

Switch between server-based and client-based messaging

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

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

Switching from server-based to client-based messaging

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

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

To switch to server-based messaging:
1. Bring down the database server.
2. Run this command: **cd $INFORMIXDIR/extend/mqblade.2.0**
3. Run this command: **rm idsmq.bld**
4. Run either of the following commands:
   - **cp idsmqc.bld idsmq.bld**
   - **ln –s idsmqc.bld idsmq.bld**

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

Related reference:

“The "informix".mqiservice table” on page 8-2
“MQCHLLIB configuration parameter” on page 9-2
“MQCHLTAB configuration parameter” on page 9-2
“MQSERVER configuration parameter” on page 9-1

Switching from client-based to server-based messaging

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

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

To switch from client-based messaging to server-based messaging:
1. Bring down the database server.

2. Run this command: **cd $INFORMIXDIR/extend/mqblade.2.0**
3. Run this command: **rm idsmq.bld**
4. Run either of the following commands:
   - **cp idsmqs.bld idsmq.bld**
   - **ln –s idsmqs.bld idsmq.bld**

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

Related reference:

"The "informix".mqiservice table" on page 8-2
"MQCHLLIB configuration parameter" on page 9-2
"MQCHLTAB configuration parameter" on page 9-2
"MQSERVER configuration parameter" on page 9-1
Chapter 9, "MQ messaging configuration parameters," on page 9-1

Verification

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

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

For more information about all of the functions used below, see Chapter 8, "MQ messaging functions," on page 8-1.

Insert data into a queue

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

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

```
BEGIN WORK;

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

(expression)          1
1 row(s) retrieved.

COMMIT WORK;
```

Read an entry from a queue

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

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

```
BEGIN WORK;

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

```
(expression) hello queue

1 row(s) retrieved.

COMMIT WORK;
```

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

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

1 row(s) inserted.

SELECT * from msgtable;

msg hello queue

1 row(s) retrieved.

COMMIT WORK;
```

Receive an entry from a queue

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

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

```
BEGIN WORK;

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

(expression)  hello queue

1 row(s) retrieved.

COMMIT WORK;
```

Publish and subscribe to a queue

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

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

Subscribe to a queue

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

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

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

BEGIN WORK;

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

```
(expression)          1

1 row(s) retrieved.

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

COMMIT WORK;
```

Unsubscribe from a queue

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

For example, specify:

```
BEGIN WORK;

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

expression)          1

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

COMMIT WORK;
```

Publish to a queue

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

For example, specify:

```
BEGIN WORK;

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

(expression)          1

COMMIT WORK;
```

Chapter 7. MQ messaging tables

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

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

Related reference:

Chapter 6, "About MQ messaging," on page 6-1

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
msgid	varchar(12)	The message ID
msgformat	varchar(8)	The message format

General table behavior

WMQ metadata tables operate in specified ways.

For every table created, the following applies:

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

Create and bind a table

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

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

```
BEGIN WORK;

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

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

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

```
BEGIN WORK;

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

For complete information about the **MQCreateVtiRead()** or **MQCreateVtiReceive()** functions, see Chapter 8, “MQ messaging functions,” on page 8-1.

Use INSERT and SELECT

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

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

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

Use a SELECT statement to display the message:

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

Retrieve the queue element

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

For example:

```
BEGIN WORK;

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

Special considerations

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

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

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

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

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

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

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

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

The above statement creates two groups of messages:
- Messages that failed the ***correlid* = 'fred'** qualification
- Messages that passed the ***correlid* = 'fred'** qualification. The one message that passed the qualification is located in the **flounder** table.

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

Table errors

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

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

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

(MQ015) - FUNCTION:MqiGetServicePolicy, SERVICE:BAD.SERVICE,
POLICY:IDS.DEFAULT.POLICY ::
BAD.SERVICE is not present in the database "informix".MQISERVICE table.
Error in line 1
Near character position 23
```

Chapter 8. MQ messaging functions

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

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

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

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

Related reference:

Chapter 6, “About MQ messaging,” on page 6-1

Service and policy tables

MQ messaging functions use three service and policy tables.

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

Table 8-1. Default policy and service values

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The "informix".mqiservice table

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

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

```
CREATE TABLE "informix".mqiservice
    servicename    LVARCHAR(256),
    queuemanager   VARCHAR(48) NOT NULL,
    queuename      VARCHAR(48) NOT NULL,
    defaultformat  VARCHAR(8) default ' ',
    ccsid          VARCHAR(6) default ' ',
    mqconnname     lvarchar(264) default '',
    mqchannelname  varchar(20) default 'SYSTEM.DEF.SVRCONN',
    mqxpt          INTEGER DEFAULT 2  CHECK ( mqxpt >= 0 AND mqxpt <= 6 ),
    mqchllib       lvarchar(512) default '',
    mqchltab       lvarchar(512) default '',
    mqserver       lvarchar(512) default '',
    PRIMARY KEY (servicename) );
```

The attributes are defined as follows:

servicename
: The service name used in the MQ functions.

queuemanager
: The queue manager service provider.

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

defaultformat
: Defines the default format.

ccsid The coded character set identifier of the destination application.

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

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

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

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

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

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

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

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

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

Related tasks:

“Switching from server-based to client-based messaging” on page 6-4
“Switching from client-based to server-based messaging” on page 6-4

Related reference:

“MQCHLLIB configuration parameter” on page 9-2
“MQCHLTAB configuration parameter” on page 9-2
“MQSERVER configuration parameter” on page 9-1
Chapter 9, “MQ messaging configuration parameters,” on page 9-1

The "informix".mqipubsub table

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

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

- Distribution list
- Receive
- Subscriber
- Publisher

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

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

The attributes are defined as follows:

pubsubname
: is the name of the publish/subscribe service.

servicebroker
: The service name of the publish/subscribe service.

receiver
: The queue on which to receive messages after subscription.

psstream
: The stream coordinating the publish/subscribe service.

pubsubtype
: The service type.

The "informix".mqipolicy table

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

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

- General
- Publish
- Receive
- Reply
- Send

- Subscribe

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

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

The attributes are defined as follows:

policyname
: The name of the policy.

messagetype
: The type of message.

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

  - The default is `Set By Queue Manager` (the queue manager sets the context).

- If set to `Pass Identity`, the identity of the request message is passed to any output messages.
- If set to `Pass All`, all the context of the request message is passed to any output messages.
- If set to `No Context`, no context is passed.

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

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

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

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

snd_retry_intrvl
: The interval (in milliseconds) between each retry.

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

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

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

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

snd_rt_exception
: Whether Exception reports are required.

snd_rt_coa
: Whether Confirm on Arrival reports are required.

snd_rt_cod
: Whether Confirm on Delivery reports are required.

snd_rt_expiry
: Whether Expiry reports are required.

reply_q
: The name of the reply queue.

reply_qmgr
: The name of the reply Queue Manager.

rcv_truncatedmsg
: Whether truncated messages are accepted.

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

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

rcv_openshared
: Whether the queue is opened as a shared queue.

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

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

pub_anonymous
: Whether the publisher registers anonymously.

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

pub_direct
: Whether the publisher should accept direct requests from subscribers.

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

pub_retain
: Whether the publication is retained by the broker.

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

sub_anonymous
: Whether the subscriber registers anonymously.

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

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

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

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

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

sub_unsuball
: Whether all topics for this subscriber are to be deregistered.

syncpoint
: Whether the operation occurred within a syncpoint.

MQCreateVtiRead() function

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

Syntax

```
>>--MQCREATEVTIREAD--(--table_name--------------------------------------->

>--+------------------------------------------------------+--)---------><
   '-,--service_name--+----------------------------------+-'
                      '-,--policy_name--+---------------+-'
                                        '-,--maxMessage-'
```

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

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

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

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

Usage

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

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

```
create table table_name (
       msg lvarchar(maxMessage),
       correlid varchar(24),
       topic varchar(40),
```

```
qname varchar(48),
msgid varchar(12),
msgformat varchar(8));
      using "informix".mq (SERVICE = service_name,
                           POLICY = policy_name,
                           ACCESS = "READ");
```

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

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

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

Table 8-2. MQCreateVtiRead() argument interpretation

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

Return codes

't' The operation was successful.

'f' The operation was unsuccessful.

Example

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

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

Insert a message into the queue:

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

Read a message from the queue:

```
select * from VtiReadTest;
```

MQCreateVtiReceive() function

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

Syntax

```
>>--MQCREATEVTIRECEIVE--(--table_name-------------------------------->

>--+-------------------------------------------------------+--)------><
   '-,--service_name--+-----------------------------------+-'
                      '-,--policy_name--+----------------+-'
                                        '-,--maxMessage--'
```

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

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

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

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

Usage

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

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

```
create table table_name (
       msg lvarchar(maxMessage),
       correlid varchar(24),
       topic varchar(40),
       qname varchar(48),
       msgid varchar(12),
       msgformat varchar(8));
```

```
using "informix".mq (SERVICE = service_name,
                     POLICY = policy_name,
                     ACCESS = "RECEIVE");
```

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

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

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

Table 8-3. MQCreateVtiReceive() argument interpretation

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

Return codes

't' The operation was successful.

'f' The operation was unsuccessful.

Example

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

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

Insert a message to the queue:

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

Read a message from the queue:

```
select * from VtiReceiveTest;
```

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

MQCreateVtiWrite() function

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

Syntax

```
>>--MQCreateVtiWrite--(--table_name--------------------------------------->

>--+------------------------------------------------+--)----------------><
   '-,--service_name--+-----------------------------+-'
                      '-,--policy_name--+----------+-'
                                        '-,--maxMessage-'
```

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

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

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

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

Usage

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

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

Table 8-4. MQCreateVtiWrite() argument interpretation

Usage	Argument interpretation
MQCreateVtiWrite(arg1)	arg1 = *table_name*

Table 8-4. MQCreateVtiWrite() argument interpretation (continued)

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

Example

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

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

MQHasMessage() function

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

Syntax

```
>>--MQHasMessage--(--+-----------------------------------------------------+--)----><
                     '-service_name--+---------------------------------+--'
                                     '-,--policy_name--+-------------+-'
                                                       '-,--correl_id-'
```

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

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

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

Usage

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

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

Table 8-5. MQHasMessage() argument interpretation

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

Return codes

1 One message or more than one message is present.

0 No Messages are available.

Error The operation was unsuccessful.

Example

This following example reads the message with the following parameters:

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

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

MQInquire() function

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

Syntax

```
>>──MQInquire──(──┬────────────────┬──selector──)──────────────────><
                  └─service_name──,─┘
```

service_name

Optional parameter. Refers to the value in the **servicename** column of the

"informix".mqiservice table. If *service_name* is not specified, IDS.DEFAULT.SERVICE is used as the service. The maximum size of *service_name* is 48 bytes.

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

Usage

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

Table 8-6. MQInquire() argument interpretation

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

You can use IBM WebSphere MQINQ() selectors.

Return codes

A string of LVARCHAR type
The operation was successful.

NULL No Messages are available.

Error The operation was unsuccessful.

Examples

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

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

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

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

The following example shows a string selector for queue depth:

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

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

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

MQPublish() function

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

Syntax

```
>>-MQPUBLISH-(--+-------------------------------------------+--msg_data----------->
                '-publisher_name-,--+---------------------+-'
                                    '-policy_name-,-------'

                                (1)
>--+---------+--+-------------+--)------------------------------------------><
   '-,-topic-'  '-,-correl_id-'
```

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 8-7. MQPublish() argument interpretation

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

Return codes

1 The operation was successful.

Error The operation was unsuccessful.

Examples

Example 1

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

This example publishes the message with the following parameters:
- *publisher_name*: default publisher
- *policy_name*: default policy
- *msg_data*: "Testing 123"
- *topic*: None
- *correl_id*: None

Example 2

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

This example publishes the message with the following parameters:
- *publisher_name*: "MYPUBLISHER"
- *policy_name*: default policy
- *msg_data*: "Testing 345"

- *topic*: "TESTTOPIC"
- *correl_id*: None

Example 3

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

This example publishes the message with the following parameters:

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

Example 4

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

This example publishes the message with the following parameters:

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

Example 5

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

This example publishes the message with the following parameters:

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

Example 6

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

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

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

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

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

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

This statement publishes the message with the following parameters:

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

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

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

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

MQPublishClob() function

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

Syntax

```
>>--MQPUBLISHCLOB--(--+-----------------------------------------+--clob_data------->
                      '-publisher_name--,--+------------------+-'
                                           '-policy_name--,---'

                                           (1)
>--+-----------+--+--------------+--)--------------------------------------------><
   '-,--topic--'  '-,--correl_id-'
```

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 8-8. 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*

Table 8-8. MQPublishClob() argument interpretation (continued)

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

Return codes

1 The operation was successful.

Error The operation was unsuccessful.

Examples

Example 1

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

This example publishes the message with the following parameters:
- *publisher_name*: default publisher
- *policy_name*: default policy
- *clob_data*: filetoclob("/work/mydata", "client")
- *topic*: None
- *correl_id*: None

Example 2

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

This example publishes the message with the following parameters:
- *publisher_name*: "MYPUBLISHER"
- *policy_name*: default policy
- *clob_data*: filetoclob("/work/mydata", "client")
- *topic*: "TESTTOPIC"
- *correl_id*: None

Example 3

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

This example publishes the message with the following parameters:

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

Example 4

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

This example publishes the message with the following parameters:

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

Example 5

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

This example publishes the message with the following parameters:

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

Example 6

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

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

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

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

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

- *topic*: "Weather"

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

This statement publishes the message with the following parameters:

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

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

This statement receives the message with the following parameters:

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

MQRead() function

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

Syntax

```
>>--MQREAD--(-+-------------------------------------------------+-)----------><
              '-service_name-+----------------------------------+-'
                             '-,--policy_name-+---------------+-'
                                              '-,--correl_id--'
```

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

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

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

Usage

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

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

Table 8-9. 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() function

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

Syntax

```
>>—MQREADCLOB—(—┬————————————————————————————————————————————————————┬—)———><
                └service_name—┬————————————————————————————————————┬┘
                              └,—policy_name—┬———————————————————┬┘
                                             └,—correl_id————————┘
```

service_name
: Optional parameter. Refers to the value in the **servicename** column of the

"informix".mqiservice table. If *service_name* is not specified, IDS.DEFAULT.SERVICE is used as the service. The maximum size of *service_name* is 48 bytes.

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

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

Usage

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

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

Table 8-10. MQReadClob() argument interpretation

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

Return codes

The contents of the message as a CLOB
: The operation was successful. If no messages are available, the result is NULL.

Error
: The operation was unsuccessful.

Example

Example 1

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

Alternatively, the following syntax can be used:

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

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

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

Example 2

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

Alternatively, the following syntax can be used:

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

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

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

Example 3

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

Alternatively, the following syntax can be used:

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

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

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

Example 4

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

Alternatively, the following syntax can be used:

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

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

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

MQReceive() function

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

Syntax

```
>>—MQRECEIVE—(——————————————————————————————————————————————)——————><
              └service_name——————————————————————————————┘
                            └,—policy_name———————————————┘
                                          └,—correl_id—┘
```

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

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

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

Usage

The **MQReceive()** function returns a message from the WMQ location specified by *service_name*, using the quality of service policy *policy_name*. This function removes the message from the queue associated with *service_name*. If *correl_id* is specified, then the first message with a matching correlation identifier is returned. If *correl_id* is not specified, then the message at the head of the queue is returned. The result of the function is a string LVARCHAR type. If no messages are available to be returned, the function returns NULL.

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

Table 8-11. MQReceive() argument interpretation

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

Table 8-11. MQReceive() argument interpretation (continued)

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

Return codes

A string of LVARCHAR type
The operation was successful.

NULL No messages are available.

Error The operation was unsuccessful.

Examples

Example 1

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

Alternatively, the following syntax can be used:

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

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

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

Example 2

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

Alternatively, the following syntax can be used:

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

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

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

Example 3

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

Alternatively, the following syntax can be used:

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

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

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

Example 4

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

Alternatively, the following syntax can be used:

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

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

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

MQReceiveClob() function

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

Syntax

```
>>--MQRECEIVECLOB--(--+------------------------------------------------------------+-->
                      '-service_name--+----------------------------------------+-'
                                      '-,--policy_name--+------------------+-'
                                                        '-,--correl_id-----'

>--)--------------------------------------------------------------------------------><
```

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

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

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

Usage

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

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

Table 8-12. 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() function

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

Syntax

```
>>──MQSEND──(──┬───────────────────────────────────────┬──────────────────►
               └─service_name──,──┬──────────────────┬─┘
                                  └─policy_name──,───┘
```

```
                                  (1)
►─msg_data──┬──────────────┬─)─────────────────────────────────────►◄
            └─,─correl_id──┘
```

Notes:

1 See the Usage section for information about 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 policy name defined by *policy_name*. If *correl_id* is specified, then the message is sent with a correlation identifier. If *correl_id* is not specified, then no correlation ID is sent with the message.

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

Table 8-13. 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*

Table 8-13. MQSend() argument interpretation (continued)

Usage	Argument interpretation
MQSend(arg1, arg2, arg3, arg4)	arg1 = *service_name* arg2 = *policy_name* arg3 = *msg_data* arg4 = *correl_id*

Return codes

1 The operation was successful.

0 or Error
The operation was unsuccessful.

Examples

Example 1

```
EXECUTE FUNCTION MQSend('Testing 123')
```

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

- *service_name*: default service name
- *policy_name*: default policy
- *msg_data*: "Testing 123"
- *correl_id*: none

Example 2

```
begin;
EXECUTE FUNCTION MQSend('MYSERVICE','Testing 901');
commit;
```

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

- *service_name*: "MYSERVICE"
- *policy_name*: default policy
- *msg_data*: "Testing 901"
- *correl_id*: none

Example 3

```
begin;
EXECUTE FUNCTION MQSend('MYSERVICE','MYPOLICY','Testing 345');
commit;
```

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

- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *msg_data*: "Testing 345"
- *correl_id*: none

Example 4

```
begin;
EXECUTE FUNCTION MQSend('MYSERVICE','MYPOLICY','Testing 678','TEST3');
commit;
```

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

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

MQSendClob() function

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

Syntax

```
►►──MQSENDCLOB──(──┬──────────────────────────────────────┬──clob_data──────────►
                   └─service_name──,──┬──────────────────┬┘
                                      └─policy_name──,───┘

                        (1)
►──┬──────────────┬──)──────────────────────────────────────────────────────────►◄
   └─,──correl_id─┘
```

Notes:

1 See the Usage section for information about 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 *service_name*, using the quality of service policy defined by *policy_name*. If *correl_id* is specified, then the message is sent with a correlation identifier. If *correl_id* is not specified, then no correlation ID is sent with the message.

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

Table 8-14. MQSendClob() argument interpretation

Usage	Argument interpretation
MQSendClob(arg1)	arg1 = *clob_data*
MQSendClob(arg1, arg2)	arg1 = *service_name* arg2 = *clob_data*
MQSendClob(arg1, arg2, arg3)	arg1 = *service_name* arg2 = *policy_name* arg3 = *clob_data*
MQSendClob(arg1, arg2, arg3, arg4)	arg1 = *service_name* arg2 = *policy_name* arg3 = *clob_data* arg4 = *correl_id*

Return codes

1 The operation was successful.

0 or Error
The operation was unsuccessful.

Examples

Example 1

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

This example sends a CLOB to the WMQ with the following parameters:
- *service_name*: default service name
- *policy_name*: default policy
- *clob_data*: filetoclob("/work/mydata", "client")
- *correl_id*: none

Example 2

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

This example sends a CLOB to the WMQ with the following parameters:
- *service_name*: "MYSERVICE"
- *policy_name*: default policy
- *msg_data*: filetoclob("/work/mydata", "client")
- *correl_id*: none

Example 3

```
begin;
EXECUTE FUNCTION MQSendClob('MYSERVICE', 'MYPOLICY',
filetoclob("/work/mydata", "client"));
commit;
```

This example sends a CLOB to the WMQ with the following parameters:
- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *msg_data*: filetoclob("/work/mydata", "client")
- *correl_id*: none

Example 4

```
begin;
EXECUTE FUNCTION MQSendClob('MYSERVICE', 'MYPOLICY',
filetoclob("/work/mydata", "client"), 'TEST3');
commit;
```

This example sends a CLOB to the WMQ with the following parameters:
- *service_name*: "MYSERVICE"
- *policy_name*: "MYPOLICY"
- *msg_data*: filetoclob("/work/mydata", "client")
- *correl_id*: "TEST3"

MQSubscribe() function

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

Syntax

```
>>--MQSUBSCRIBE--(--+----------------------------------------------+--topic--)---------><
                    '--subscriber_name--,--+--------------------+--'
                                           '--policy_name--,--'
```

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 8-15. MQSubscribe() argument interpretation

Usage	Argument interpretation
MQSubscribe(arg1)	arg1 = *topic*
MQSubscribe(arg1, arg2)	arg1 = *service_name* arg2 = *topic*
MQSubscribe(arg1, arg2, arg3)	arg1 = *service_name* arg2 = *policy_name* arg3 = *topic*

Return codes

1 The operation was successful.

Error The operation was unsuccessful.

Examples

Example 1

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

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

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

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

- *subscriber_name*: "IDS.DEFAULT.SUBSCRIBER"

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

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

This statement publishes the message with the following parameters:

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

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

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

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

Example 2

```
begin;
EXECUTE FUNCTION MQSubscribe('Weather');
commit;
```

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

- *subscriber_name*: default subscriber
- *policy_name*: default policy
- *topic*: "Weather"

Example 3

```
begin;
EXECUTE FUNCTION MQSubscribe('PORTFOLIO-UPDATES',
 'BASIC-POLICY', 'Stocks:Bonds');
commit;
```

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

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

MQTrace() function

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

Syntax

```
►►—MQTRACE—(—trace_level—,—trace_file—)————————————————►◄
```

trace_level
: Required parameter. Integer value specifying the trace level, currently only a value of greater than 50 results in output.

trace_file
: Required parameter. The full path and name of the file to which trace information is appended. The file must be writable by user **informix**.

To enable tracing, you must first create a trace class by inserting a record into the **systemtraceclasses** system catalog:

```
insert into informix.systraceclasses(name) values ('idsmq')
```

For more details regarding tracing, see the *IBM Informix Guide to SQL: Reference*.

Example

Enable tracing at a level of 50 with an output file of `/tmp/trace.log`:

```
EXECUTE PROCEDURE MQTrace(50, '/tmp/trace.log');
```

Execute a request:

```
begin;
EXECUTE FUNCTION MQSend('IDS');
commit;
```

Look at the trace output:

```
14:19:38 Trace ON level : 50
14:19:47  >>ENTER : mqSend<<
14:19:47    status:corrid is null
14:19:47  >>ENTER : MqOpen<<
14:19:47   status:MqOpen @ build_get_mq_cache()
14:19:47  >>ENTER : build_get_mq_cache<<
14:19:47   status:build_get_mq_cache @ mi_get_database_info()
14:19:47   status:build_get_mq_cache @ build_mq_service_cache()
14:19:47  >>ENTER : build_mq_service_cache<<
14:19:47  <<EXIT : build_mq_service_cache>>
14:19:47   status:build_get_mq_cache @ build_mq_policy_cache()
14:19:47  >>ENTER : build_mq_policy_cache<<
14:19:47  <<EXIT : build_mq_policy_cache>>
14:19:47   status:build_get_mq_cache @ build_mq_pubsub_cache()
14:19:47  >>ENTER : build_mq_pubsub_cache<<
14:19:47  <<EXIT : build_mq_pubsub_cache>>
14:19:47  <<EXIT : build_get_mq_cache>>
14:19:47   status:MqOpen @ MqiGetServicePolicy()
14:19:47  >>ENTER : MqiGetServicePolicy<<
14:19:47  <<EXIT : MqiGetServicePolicy>>
14:19:47    MQI:MqOpen @ MQCONNX()
14:19:47   status:MqOpen @ MqXadsRegister()
14:19:47  >>ENTER : MqXadsRegister<<
14:19:47   status:MqXadsRegister @ ax_reg()
14:19:47  <<EXIT : MqXadsRegister>>
14:19:47   status:MqOpen @ MqGetMqiContext()
14:19:47  >>ENTER : MqGetMqiContext<<
14:19:47    MQI:MqGetMqiContext @ MQOPEN()
14:19:47  <<EXIT : MqGetMqiContext>>
14:19:47  <<EXIT : MqOpen>>
14:19:47  >>ENTER : MqTransmit<<
14:19:47  >>ENTER : MqBuildMQPMO<<
14:19:47  <<EXIT : MqBuildMQPMO>>
14:19:47  >>ENTER : MqBuildMQMDSend<<
14:19:47  <<EXIT : MqBuildMQMDSend>>
14:19:47    MQI:MqTransmit @ MQPUT()
14:19:47  <<EXIT : MqTransmit>>
```

```
14:19:47  <<EXIT : mqSend>>
14:19:47  >>ENTER : MqEndTran<<
14:19:47    MQI:MqEndTran @ MQCMIT()
14:19:47   status:MqEndTran @ MqShut()
14:19:47  >>ENTER : MqShut<<
14:19:47   status:MqEndTran @ MQDISC
14:19:47  <<EXIT : MqEndTran>>:
```

MQUnsubscribe() function

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

Syntax

```
>>──MQUNSUBSCRIBE──(──┬──────────────────────────────────────────────┬──topic──)──────────><
                      └─subscriber_name──,──┬─────────────────────┬──┘
                                            └─policy_name──,──────┘
```

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 8-16. MQUnsubscribe() argument interpretation

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

Table 8-16. 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() function

The **MQVersion()** function returns version information.

The **MQVersion()** function returns the version of the MQ messaging extension.

Syntax

```
►►──MQVersion──()──────────────────────────────────────────────────►◄
```

Example

Show the version:

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

Chapter 9. MQ messaging configuration parameters

When you use MQ messaging over a network, you must set several database server configuration parameters,

These configuration parameters correspond to IBM WebSphere MQ (WMQ) environment variables.

Related tasks:

“Switching from client-based to server-based messaging” on page 6-4

Related reference:

Chapter 6, “About MQ messaging,” on page 6-1

“The "informix".mqiservice table” on page 8-2

MQSERVER configuration parameter

Use the MQSERVER configuration parameter to define a channel, specify the location of the IBM WebSphere MQ server, and specify the communication method to be used.

`onconfig.std` *value*
: none

range of values
: *ChannelName/TransportType/ConnectionName*

takes effect
: When the database server is stopped and restarted

Usage

You must set this configuration parameter when you use MQ messaging over a network. The configuration parameter contains the same information as the same as the WMQ MQSERVER environment variable.

The connection name must be a fully qualified network name.

The connection and channel names cannot include contain the forward slash (/) character, because it is used to separate the channel name, transport type, and connection name.

For more information about channel and connection names and transport types, see your IBM WebSphere MQ documentation.

Related tasks:

“Switching from server-based to client-based messaging” on page 6-4

“Switching from client-based to server-based messaging” on page 6-4

Related reference:

“The "informix".mqiservice table” on page 8-2

“MQCHLLIB configuration parameter” on page 9-2

“MQCHLTAB configuration parameter” on page 9-2

MQCHLLIB configuration parameter

Use the MQCHLLIB configuration parameter to specify the path to the directory containing the client channel definition table.

onconfig.std *value*
: none

range of values
: complete path name

takes effect
: When the database server is stopped and restarted

Usage

You must set this configuration parameter when you use MQ messaging over a network. The configuration parameter contains the same information as the same as the IBM WebSphere MQ (WMQ) MQCHLLIB environment variable.

For example, if the path is `/var/mqm`, specify:

```
MQCHLLIB /var/mqm
```

Related tasks:

"Switching from server-based to client-based messaging" on page 6-4

"Switching from client-based to server-based messaging" on page 6-4

Related reference:

"The "informix".mqiservice table" on page 8-2

"MQCHLTAB configuration parameter"

"MQSERVER configuration parameter" on page 9-1

MQCHLTAB configuration parameter

Use the MQCHLTAB configuration parameter to specify the name of the client channel definition table.

onconfig.std *value*
: none

range of values
: String for the file name

takes effect
: When the database server is stopped and restarted

Usage

You must set this configuration parameter when you use MQ messaging over a network. The configuration parameter contains the same information as the same as the IBM WebSphere MQ (WMQ) MQCHLTAB environment variable.

The default file name in the WMQ MQCHLTAB environment variable is `AMQCLCHL.TAB`.

For example, if the name of the client channel definition table that you are using is `CCD1`, specify:

```
MQCHLTAB CCD1.TAB
```

Related tasks:

"Switching from server-based to client-based messaging" on page 6-4

"Switching from client-based to server-based messaging" on page 6-4

Related reference:

"The "informix".mqiservice table" on page 8-2

"MQCHLLIB configuration parameter" on page 9-2

"MQSERVER configuration parameter" on page 9-1

Chapter 10. MQ messaging error handling

This topic describes MQ messaging error codes.

SQL State	Description
MQ000	Memory allocation failure in %FUNC%.
MQPOL	MQOPEN Policy : %POLICY%
MQSES	MQOPEN Session : %SESSION%
MQRCV	Read %BYTES% from the queue.
MQNMS	No data read/received, queue empty.
MQSUB	Subscribing to %SUBSCRIBE%.
MQVNV	VTI Table definition parameter NAME:%NAME% VALUE:%VALUE%.
MQNPL	VTI No policy defined for table mapped to MQ. Must define table with policy attribute.
MQNSV	VTI No service defined for table mapped to MQ. Must define table with service attribute.
MQNAC	VTI No access defined for table mapped to MQ. Must define table with access attribute.
MQBAC	VTI Invalid Access specification FOUND:%VALUE%, possible values%VALONE% or %VALTWO%.
MQVCN	VTI Qualified : Column 'correlid' cannot be qualified with NULL.
MQVTB	Table missing required 'message' column. Message column is bound to the queue, it is mandatory.
MQVSP	VTI mapped Queue did not include the POLICY and SESSION columns.
MQVIA	VTI table definition invalid access type (%VALUE%), valid access types are %READ% or %RECEIVE%.
MQVMS	VTI mapped queue missing SERVICE specification.
MQVMA	VTI mapped QUEUE creation did not include ACCESS definition.
MQVMP	VTI mapped QUEUE creation did not include POLICY specification.
MQVQC	VTI queue mapping, Column '%COLUMN%' must be qualified with a constant.
MQVQN	VTI queue mapping, Column '%COLUMN%' cannot be qualified with NULL.
MQVQE	VTI queue mapping, Column '%COLUMN%' can only use equality operator.
MQVQF	VTI queue mapping, column '%COLUMN%' - failed to fetch field.
MQSUN	Invalid selector '%IDX%' found, path not possible.
MQERX	Extended error : '%FUNC%', code:%CODE% explain: %EXPLAIN%, refer to MQSeries publication for further description.
MQGEN	%FUNC% encountered error %ERR% with accompanying message : %MSG%
MQTNL	Topic cannot be NULL.
MQCNL	Internal error encountered NULL context.
MQNLM	Cannot send NULL message.
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

SQL State	Description
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""

Chapter 11. Sample MQ messaging code

This topic contains sample SQL statements that you can run in the **stores_demo** database, using DB-Access.

The sample statements are for one queue manager. However, you can use multiple queue managers.

```
begin;

select MQSEND ('lser.qm1', 'IDS.DEFAULT.POLICY',
       TRIM(fname)    || '  ' || TRIM(lname) || '|'
       || TRIM(company)  || ', ' || TRIM(address1) || ', '
       || TRIM(NVL(address2,"")) || ', ' || TRIM(city) || ', '
       || state || ', ' || zipcode || '|' || TRIM(phone) || '|' ,
      state)
     from customer;

select MQSEND ('lser.qm1', 'IDS.DEFAULT.POLICY',
       stock_num || '|' || manu_code || '|' || TRIM(description)
       || '|' || unit_price || '|' || unit
       || '|' || TRIM(unit_descr) || '|' ,
       manu_code)
     from stock;
commit;


select first 3 MQREAD('lser.qm1') from systables;
begin;
execute function MQREAD('lser.qm1','IDS.DEFAULT.POLICY','AZ');
rollback;

begin;
execute function MQREAD ('lser.qm1');
execute function MQREAD ('lser.qm1');
execute function MQRECEIVE ('lser.qm1');
execute function MQRECEIVE ('lser.qm1');
rollback;

begin;
select first 5 MQREAD ('lser.qm1') from systables;
select first 5 MQREAD ('lser.qm1') from systables;
select first 1 MQRECEIVE ('lser.qm1','IDS.DEFAULT.POLICY','AZ')
 from systables;
select first 1 MQRECEIVE ('lser.qm1','IDS.DEFAULT.POLICY','HSK')
 from systables;
rollback;

begin;
select first 5 MQREAD ('lser.qm1') from systables;
select first 5 MQREAD ('lser.qm1') from systables;
select first 1 MQRECEIVE ('lser.qm1','IDS.DEFAULT.POLICY','AZ')
 from systables;
select first 1 MQRECEIVE ('lser.qm1','IDS.DEFAULT.POLICY','HSK')
 from systables;
commit;

execute function mqinquire('lser.qm1',20);
execute function mqinquire('lser.qm1',"MQIA_Q_TYPE");
execute function mqinquire('lser.qm1',3);
execute function mqinquire('lser.qm1',"MQIA_CURRENT_Q_DEPTH");
execute function mqhasmessage('lser.qm1');
execute function mqhasmessage('lser.qm1','IDS.DEFAULT.POLICY','CA');
```

```
execute function mqhasmessage('lser.qm1','IDS.DEFAULT.POLICY','XY');

execute function MQCreateVtiRead("qm0vti", "lser.qm1");
execute function MQCreateVtiReceive("qm0vtir", "lser.qm1");
execute function MQCreateVtiWrite("qm0vtiw", "lser.qm1");
execute function MQCreateVtiReceive("qm1vti", "lser.qm1");

insert into qm0vtiw(msg) values ("Informix Dynamic Server");
begin;

select skip 10 first 5 * from qm0vtir;
select * from qm1vti;
insert into qm1vti(msg) values ("Informix Dynamic Server");
select * from qm1vti;
commit;
```

Part 3. Binary data types

The binary18 and binaryvar data types allow you to store binary-encoded strings, which can be indexed for quick retrieval.

You can use string manipulation functions to validate the data types and bitwise operation functions that allow you to perform bitwise logical AND, OR, XOR comparisons or apply a bitwise logical NOT to a string.

Because the binary data types are unstructured types, they can store many different types of information, for example, IP addresses, MAC addresses, or device identification numbers from RFID tags. The binary data types can also store encrypted data in binary format, which saves disk space. Instead of storing an IP address like *xxx.xxx.xxx.xxx* as a CHAR(15) data type, you can store it as a binaryvar data type, which uses only 6 bytes.

Chapter 12. Binary data types overview

To implement binary data types, the Scheduler must be running and the database must conform to requirements. The binary18 and binaryvar data types have certain restrictions due to the nature of binary data.

The Scheduler must be running in the database server. If the Scheduler is not running when you create a binary data type, a message that the data type is not found is returned.

The database that contains the binary data types must meet the following requirements:

- The database must be logged.
- The database must not be defined as an ANSI database.

If you attempt to create a binary data type in an unlogged or ANSI database, the message `DataBlade registration failed` is printed in the online message log.

Binary data type can be used in the following situations:

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

Binary data types have the following limitations:

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

Chapter 13. Store and index binary data

This chapter describes the binary data types and how to insert and index binary data.

Binary data types

You can store and index binary data by using the binaryvar and binary18 data types.

The binaryvar data type

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

The binary18 data type

The binary18 data type is a fixed-length opaque data type that holds 18 bytes. Input strings shorter than 18 bytes are right-padded with zeros (`00`). Strings longer than 18 bytes are truncated.

The binary18 data type has the advantage of not having its length stored as part of the byte stream. When inserting data into the binaryvar data type, the first byte must be the length of the byte array. The binary18 data type does not have this restriction.

ASCII representation of binary data types

Binary data types are input using a 2-digit ASCII representation of the characters in the hexadecimal range of 0-9, A-F. The characters A-F are not case-sensitive and you can add a leading **0x** prefix to the string. You must enter an even number of bytes up to the maximum number of encoded bytes permitted, otherwise an error is generated. For example, 36 bytes are input to represent the binary18 data type. No spaces or other separators are supported.

Each 2-byte increment of the input string is stored as a single byte. For example, the 2-byte ASCII representation of "AB" in hexadecimal notation is divided into blocks of four binary characters, where `1010 1011` equals one byte.

Binary data type examples

Example 1: binaryvar data type

The following code stores the binary string of `0123456789` on disk:

```
CREATE TABLE  bindata_test (int_col integer, bin_col binaryvar)

INSERT INTO bindata_test values (1, '30313233343536373839')
INSERT INTO bindata_test values (2, '0X30313233343536373839')
```

Example 2: binary18 data type

The following code inserts the string IBMCORPORATION2006:

```
CREATE TABLE bindata_test (int_col integer, bin_col binary18)

INSERT INTO bindata_test values (1,'49424d434f52504f524154494f4e32303036')
INSERT INTO bindata_test values (2,'0x49424d434f52504f524154494f3e32303036')
```

Insert binary data

You can use one of two methods to insert binary data with the binary data types: an SQL INSERT statement that uses the ASCII representation of the binary data type or an SQL INSERT statement from a Java™ or C program that treats the column as a byte stream. For example, given the following table:

```
CREATE TABLE network_table (
mac_address binaryvar NOT NULL,
device_name varchar(128),
device_location varchar(128),
device_ip_address binaryvar,
date_purchased date,
last_serviced date)
```

Using an SQL INSERT statement that uses the ASCII representation of the binaryvar or binary18 column:

```
INSERT INTO network_table VALUES ( '000012DF4F6C', 'Network Router 1',
'Basement',  'C0A80042', '01/01/2001', '01/01/2006');
```

Using an SQL INSERT statement from a Java program that treats the column as a byte stream, such as the JDBC **setBytes()** method:

```
String binsqlstmt = "INSERT INTO network_table (mac_address, device_name,
device_location, device_ip_address) VALUES ( ?, ?, ?, ? );
PreparedStatement stmt = null;
byte[] maddr = new byte[6];
byte[] ipaddr = new byte[4];
try
{
   stmt = conn.prepareStatement(binsqlstmt);
   maddr[0] = 0;
    maddr[1] = 0;
   maddr[2] = 18;
   maddr[3] = -33;
   maddr[4] = 79;
   maddr[5] = 108;
   stmt.setBytes(1, maddr);
   stmt.setString(2, "Network Router 1");
   stmt.setString(3, "Basement");
   ipaddr[0] = -64;
   ipaddr[1] = -88;
   ipaddr[2] = 0;
   ipaddr[3] = 66;
   stmt.setBytes(4,ipaddr);
   stmt.executeUpdate();
   stmt.close()
}
catch
{
   System.out.println("Exception: " + e);
   e.printStackTrace(System.out);
   throw e;
}
```

Index binary data

The binaryvar and binary18 data types support indexing using the B-tree access method for single-column indexes and composite indexes. Nested-loop join operations are also supported.

For example, given the following table:

```
CREATE TABLE network_table (
mac_address binaryvar NOT NULL,
device_name varchar(128),
device_location varchar(128),
device_ip_address binaryvar,
date_purchased date,
last_serviced date)
```

The following statement can be used to create the index:

```
CREATE UNIQUE INDEX netmac_pk ON network_table (mac_address) USING btree;
```

Chapter 14. Binary data type functions

This chapter describes functions for the binary data types and provides detailed information about each function's syntax and usage.

Bitwise operation functions

These functions perform bitwise operations on binary18 or binaryvar fields. The expressions can be either binary18 or binaryvar columns or they can be expressions that have been implicitly or explicitly cast to either the binary18 or the binaryvar data type.

The return type for all of these functions is either the binary18 or the binaryvar data type.

The bit_and() function

The **bit_and()** function performs a bitwise logical AND operation on two binary data type columns.

Syntax

bit_and(*column1*, *column2*)

column1, *column2*
Two input binary data type columns.

Usage

If the columns are different lengths, the return value is the same length as the longer input parameter with the logical AND operation performed up to the length of the shorter parameter.

Return codes

The function returns the value of the bitwise logical AND operation.

If either parameter is NULL, the return value is also NULL.

Example

In the following example, the value of `binaryvar_col1` is '00086000'.

```
SELECT bit_and(binaryvar_col1, '0003C000'::binaryvar) FROM table WHERE x = 1
expression
------------------
00004000
```

The bit_complement() function

The **bit_complement()** function performs a logical NOT, or *one's complement* on a single binary data type column.

Syntax

bit_complement(*column*)

column The input binary data type column.

Usage

The function changes each binary digit to its complement. Each `0` becomes a `1` and each `1` becomes a `0`.

Return codes

The function returns the value of the bitwise logical NOT operation.

Example

In the following example the value of `binaryvarcol1` is '00086000':

```
SELECT bit_complement(binaryvar_col1) FROM table WHERE x = 1
expression
------------------
FFF79FFF
```

The bit_or() function

The **bit_or()** function performs a bitwise logical OR on two binary data type columns.

Syntax

```
bit_or(column1, column2)
```

column1, *column2*
: Two input binary data type columns.

Usage

If the columns are of different length, the return value is the same length as the longer input parameter, with the OR operation performed up to the length of the shorter parameter. The remainder of the return value is the unprocessed data in the longer string.

Return codes

The function returns the value of the bitwise logical OR operation.

If either parameter is NULL, the return value is also NULL.

Example

In the following example, the value `binaryvarcol1` is '00006000':

```
SELECT bit_or(binaryvar_col1, '00080000'::binaryvar) FROM table WHERE x = 1
expression
------------------
00086000
```

The bit_xor() function

The **bit_xor()** function performs a bitwise logical XOR on two binary data type columns.

Syntax

```
bit_xor(column1, column2)
```

column1, *column2*
Two input binary data type columns.

Usage

If the columns are of different lengths, the return value is the same length as the longer input parameter, with the XOR operation performed up to the length of the shorter parameter. The remainder of the return value is the unprocessed data in the longer parameter.

Return codes

The function returns the value of the bitwise logical XOR operation.

If either parameter is NULL, the return value is also NULL.

Example

In the following example, the value of `binaryvarcol1` is '00086000':

```
SELECT bit_xor(binaryvar_col1, '00004000'::binaryvar) FROM table WHERE x = 1'
expression
------------------
00082000
```

Support functions for binary data types

Supporting functions for binary data types include the SQL **LENGTH()** and **OCTET_LENGTH()** functions that allow you to determine the length of a column. The **bdttrace()** function is used to trace events related to using binary data types.

The bdtrelease() function

The **bdtrelease()** function provides the version number of the binary data types.

Syntax

```
bdtrelease(void)
```

Usage

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

Return codes

This function returns the name and version number of the binary data types.

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
```

The bdttrace() function

The **bdttrace()** function specifies the location where the trace file is written.

Syntax

```
bdttrace(filename)
```

filename
: The full path and name of the file to which trace information is appended. The file must be writable by user **informix**. If no file name is provided, a standard `session_id.trc` file is placed in the `$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 binary data types.

To enable tracing, create a trace class by inserting a record into the **systemtraceclasses** system catalog:

```
insert into informix.systraceclasses(name) values ('binaryUDT')
```

For more details regarding tracing, see the *IBM Informix Guide to SQL: Reference*.

Example

```
bdttrace(tracefile)
```

The LENGTH() function

Use the **LENGTH()** SQL function to determine if the string is from a binaryvar or a binary18 column. The **LENGTH()** function returns the number of bytes in a column.

Syntax

```
LENGTH(column)
```

column The binary data type column.

Usage

This function returns the length of the column in bytes as an integer. For the binary18 data type, the function always returns 18.

For binary data types, the SQL **LENGTH()** and **OCTET_LENGTH()** functions return the same value. For more information about length functions, see the *IBM Informix Guide to SQL: Reference*.

Example

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

The OCTET_LENGTH() function

Use the **OCTET_LENGTH()** SQL function to determine if the string is from a binaryvar or a binary18 column. The **OCTET_LENGTH()** function returns the number of octets (bytes).

Syntax

```
OCTET_LENGTH(column)
```

column The binary data type column.

Usage

This function returns the length of the column in bytes as an integer. For the binary18 data type, the function always returns 18.

For binary data types, the SQL **LENGTH()** and **OCTET_LENGTH()** functions return the same value. For more information about length functions, see the *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

You can perform basic text searching for words and phrases in a document repository stored in a column of a table.

In traditional relational database systems, you must use a LIKE or MATCHES condition to search for text data and use the database server to perform the search. IBM Informix uses the open source CLucene text search package to perform basic text searches. This text search package and its associated functions, known as the text search *engine*, is specifically designed to perform fast retrieval and automatic indexing of text data. The text search engine runs in virtual processors that are controlled by the database server.

To perform basic text searches, you create a **bts** index on one or more text columns and then use the **bts_contains()** search predicate function to query the text data.

You can configure how to index the text data by specifying an analyzer. Each analyzer uses different criteria to index the data. By default the Standard analyzer is used.

You can specify synonyms for data that has multiple words for the same information, for example, proper names with multiple spellings. You can use canonical mapping to create a static list of synonyms. You can create a thesaurus with synonyms that you can update dynamically.

To search for words and phrases you use a predicate called **bts_contains()** that instructs the database server to call the text search engine to perform the search.

For example, to search for the string `century` in the column **brands** in the table **products** you use the following statement:

```
SELECT id FROM products
WHERE bts_contains(brands, 'century');
```

The search predicate takes a variety of arguments to make the search more detailed than one using a LIKE condition. Search strategies include single and multiple character wildcard searches, fuzzy and proximity searches, AND, OR and NOT Boolean operations, range options, and term-boosting.

If you store XML, JSON, or BSON documents, you can create customized structured indexes so that you can search columns by XML tags, attributes, and paths, or JSON fields, values, and paths. Customize the index with XML or JSON index parameters.

You can use basic text search functions to perform maintenance tasks, such as compacting the **bts** index and obtaining the list of indexed field names.

Chapter 15. Preparing for basic text searching

Before you can perform basic text searching, you must prepare the server environment and create the **bts** index. Review the requirements and restrictions.

To prepare for basic text searching, complete these tasks:
1. Create a default sbspace.
2. Optional: Create an sbspace for the **bts** index.
3. Optional: Create a space for temporary data.
4. Create the **bts** index.

Basic text search functions run in a BTS virtual processor, which means that only one query or other type of index operation runs at a time in each virtual processor. When you create a bts index, the BTS virtual processor class is created automatically.

Basic text search requirements and restrictions

When you plan how to configure basic text searching, you must understand the requirements and restrictions.

Database server requirement

The Scheduler must be running in the database server. If the Scheduler is not running when you create a **bts** index, a message that the access method is not found is returned.

Database requirements

The database that contains the **bts** index must be logged and must not be an ANSI database. If you attempt to create a **bts** index in an unlogged or ANSI database, the message `DataBlade registration failed` is printed in the database server message log.

Data type support

To use basic text searching, you must store the text data in a column of data type BLOB, BSON, CHAR, CLOB, JSON, LVARCHAR, NCHAR, NVARCHAR, or VARCHAR.

Although you can store searchable text in a column of the BLOB data type, you cannot create a basic text search index on binary data. BLOB data type columns must contain text.

Locales and languages support

Basic text search queries can use most multi-byte character sets and global language support, including UTF-8, and can use ideographic languages such as Chinese, Korean, and Japanese if you specify the CJK analyzer.

JSON or BSON documents must be in a database with a UTF-8 locale.

High availability support

You can run basic text search queries on primary and all types of secondary servers in high-availability clusters.

Index characteristics restrictions

The following characteristics are not supported for **bts** indexes:
- Fill factors
- Index clustering
- Unique indexes

Indexed document restrictions

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

The size of a document that you want to index is limited by the amount of available virtual memory on your machine. For example, if you have 1 GB of available virtual memory, you can only index documents that are smaller than 1 GB.

Query restrictions

You cannot include basic text search queries in distributed queries or parallel database queries.

Creating a default sbspace

A default sbspace must exist before you create a **bts** index. The database server sets up internal directories for basic text searching in a default sbspace.

The database server also stores **bts** indexes in the default sbspace unless you explicitly specify another sbspace when you create the index. Be sure the default sbspace is large enough to hold all of these objects. Monitor the size of the default sbspace and increase its size when necessary.

If you do not explicitly create a default sbspace and set the SBSPACENAME configuration parameter in the `onconfig` file before you create a **bts** index, the database server creates a default sbspace automatically before running the CREATE INDEX statement, according to the following criteria in this order:
- If storage provisioning is configured, the default sbspace is created in the designated storage pool.
- If the root dbspace was created in a directory, the default sbspace is created in the same directory and could use the same files system as the root dbspace.
- If the root dbspace is a raw device in the `/dev` directory, the default sbspace is created in the `$INFORMIXDIR/tmp` directory.

The sbspace for **bts** index must have buffering enabled. Buffering is enabled by default when you create an sbspace. You can use various methods to create an sbspace, including the **onspaces** utility, the SQL administration API **task()** function with the **`create sbspace`** argument, or through storage provisioning, if you have configured a storage pool.

To create the default sbspace:
1. Set the SBSPACENAME configuration parameter in the configuration file to the name of your default sbspace.

   The following example sets the name of the default sbspace to **sbsp1**:
   ```
   SBSPACENAME sbsp1
   ```
2. Restart the database server.
3. Create the sbspace.

The following example creates an sbspace called **sbsp1** in the file `c:\IFMXDATA\sbspace` by using the **onspaces** utility:

```
onspaces -c -S sbsp1 -p c:\IFMXDATA\sbspace -o 0 -s 100000
```

Related reference:

SBSPACENAME configuration parameter (Administrator's Reference)

onspaces -c -S: Create an sbspace (Administrator's Reference)

onspaces -c -x: Create an extspace (Administrator's Reference)

create sbspace argument: Create an sbspace (SQL administration API) (Administrator's Reference)

create sbspace from storagepool argument: Create an sbspace from the storage pool (SQL administration API) (Administrator's Reference)

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 database server stores **bts** indexes in the default sbspace.

In general, the sbspace for a **bts** index should be at least the size of the data being indexed. A highly optimized index might take up to three times the size of the data being indexed.

The sbspace for **bts** index must have buffering enabled. Buffering is enabled by default when you create an sbspace. You can use various methods to create an sbspace, including the **onspaces** utility, the SQL administration API **task()** function with the **`create sbspace`** argument, or through storage provisioning, if you have configured a storage pool.

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

```
onspaces -c -S bts_sbspace -o 0 -s 100000 -p /dev/sbspace
```

To create an extspace:

1. Create a directory for the index.
2. Create the extspace by using the **onspaces** utility.

The following example creates a directory and an extspace:

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

Related reference:

onspaces -c -S: Create an sbspace (Administrator's Reference)

onspaces -c -x: Create an extspace (Administrator's Reference)

create sbspace argument: Create an sbspace (SQL administration API) (Administrator's Reference)

create sbspace from storagepool argument: Create an sbspace from the storage pool (SQL administration API) (Administrator's Reference)

Creating a space for temporary data

Basic text searching creates temporary data while processing **bts** indexes. You can create a separate space for temporary data and specify it when you create the **bts** index.

For best performance, the space should be a temporary sbspace since data and metadata for temporary files are not logged. However, you can also use an sbspace or an extspace.

If you do not specify a separate space for temporary data when you create the **bts** index with the **tempspace** index parameter, the database server stores temporary data in one of the following locations, according to the criteria in the following order:

- The sbspace specified in the CREATE INDEX statement.
- A temporary sbspace that is specified by the SBSPACETEMP configuration parameter. The temporary sbspace with the most free space is used. If no temporary sbspaces are listed, the sbspace with the most free space is used.
- If the SBSPACETEMP configuration parameter is not set and you have a storage pool that is set up, a temporary sbspace is created and the SBSPACETEMP configuration parameter is set dynamically in the `onconfig` file.
- The sbspace specified by the SBSPACENAME configuration parameter.

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
```

Alternatively, you could create a temporary sbspace through storage provisioning, if you have configured a storage pool.

Related reference:

Chapter 21, “Basic text search performance,” on page 21-1

onspaces -c -S: Create an sbspace (Administrator's Reference)

create tempsbspace from storagepool argument: Create a temporary sbspace from the storage pool (SQL administration API) (Administrator's Reference)

Creating a bts index

You create a **bts** index by using the **bts** access method and specifying index parameters and other options.

Before you create a **bts** index, plan which index parameters and other options you want to use.

To create a **bts** index:

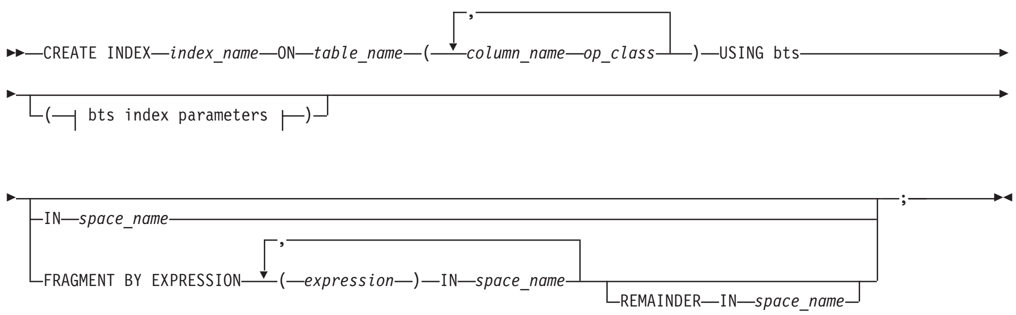
1. Complete prerequisite tasks that are necessary for the index parameters that you plan to include for the index. For example, many index parameters use tables or files that you must create before you create the index.
2. Create an index by specifying the **bts** access method.

bts access method syntax

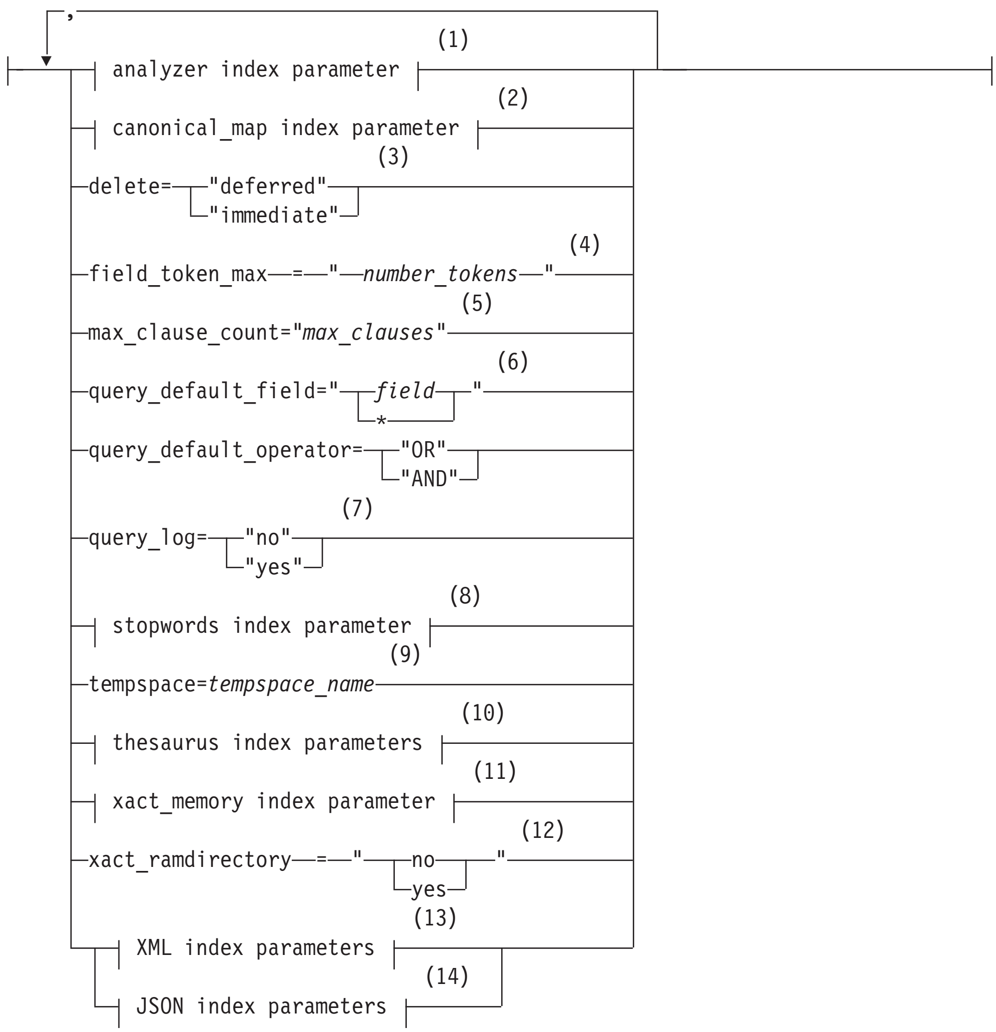
The **bts** access method is a secondary access method to create indexes that support basic text search queries.

Instead of using the **bts** access method to create a **bts** index, you can run the Informix JSON **createTextIndex** command. Use the same syntax for **bts** index parameters for both methods.

Syntax



bts index parameters:



Notes:

1 See "analyzer index parameter" on page 19-1.

2 See "canonical_maps index parameter" on page 15-8.

3 See "delete index parameter" on page 15-11.

4 See "field_token_max index parameter" on page 15-12.

5 See "max_clause_count index parameter" on page 15-12.

6 See "query_default_field index parameter" on page 15-13.

7 See "query_log index parameter" on page 15-14.

8 See "stopwords index parameter" on page 15-15.

9 See "Creating a space for temporary data" on page 15-4.

10 See "thesaurus index parameters" on page 15-17.

11 See "xact_memory index parameter" on page 15-18.

12 See "xact_ramdirectory index parameter" on page 15-19.

13 See "XML index parameters syntax" on page 18-1.

14 See "JSON index parameters syntax" on page 17-2.

Element	Description
column_name	The name of the column in the table that contains the text documents to search.
expression	The expression that defines an index fragment. The expression must return a Boolean value. The expression can contain only columns from the current table and data values from only a single row. The expression cannot include the following elements: • Subqueries • Aggregates are not allowed. T • The built-in CURRENT, DATE, SYSDATE, and TODAY functions • The **bts_contains()** search predicate For more information about expressions, see Expression.
field	The name of the field to set as the default field in basic text search queries instead of the `contents` field.
index_name	The name of the **bts** index.
max_clauses	The maximum number of clauses in a basic text search query. Default is 1024.
number_tokens	The maximum number of tokens to index for each document. Default is 10 000. Maximum is 2 000 000 000.
op_class	The operator class for the data type that is specified in the *column_name* element.
space_name	The name of the sbspace or extspace in which to store the **bts** index.
table_name	The name of the table for which you are creating the index.
tempspace_name	The name of the space in which to store temporary files.

Usage

Include a comma between index parameters.

You must create a **bts** index for each text column that you plan to search. You can either create a separate **bts** index for each text column, or create a composite index on multiple text columns in a table by including multiple column and operator class pairs. You cannot create a composite index that includes a JSON or BSON column. If you want to index each column separately, include the **query_default_field="*"** index parameter.

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

When you create a **bts** index, you specify the operator class that is defined for the data type of the column that is indexed. An operator class is a set of functions that the database server associates with the **bts** access method to optimize queries and build indexes. Each of the data types that support a **bts** index has a corresponding operator class. The following table lists each data type and its corresponding operator class.

Table 15-1. Data types and the corresponding operator classes

Data type	Operator class
BLOB	bts_blob_ops
BSON	bts_bson_ops
CHAR	bts_char_ops
CLOB	bts_clob_ops
JSON	bts_json_ops
LVARCHAR	bts_lvarchar_ops
NCHAR	bts_nchar_ops
NVARCHAR	bts_nvarchar_ops
VARCHAR	bts_varchar_ops

Examples

Example 1: Create a bts index and store it in an sbspace

For example, suppose that your search data is contained in a column that is named **brands**, of data type CHAR, in a **products** table. To create a **bts** index that is named **desc_idx** in the sbspace **sbsp1**, use the following syntax:

```
CREATE INDEX desc_idx ON products (brands  bts_char_ops)
 USING bts IN sbsp1;
```

Example 2: Create a fragmented bts index

The following example stores the **bts_idx** index in three sbspaces by fragmenting the index according to an expression:

```
CREATE INDEX bts_idx ON bts_tab(col2 bts_char_ops) USING bts
      FRAGMENT BY EXPRESSION
        (col1 <= 1000000) IN bts_sbspace00,
        (col1 > 1000000 and col1 <= 2000000)
        IN bts_sbspace01,
        REMAINDER IN bts_sbspace36;
```

Related reference:

Informix JSON commands (JSON compatibility)

canonical_maps index parameter

You can index synonyms by creating a canonical map. You specify canonical mapping strings with the **canonical_maps** index parameter when you create the **bts** index.

canonical_maps index parameter:

```
                      ┌ , ─────────────────────────────────────────────────┐
                      ▼                                                    │
├──canonical_maps──=──"──┬─(───┬─{original_char}──────────────────┬─:─{mapped_string}─┴─)─┬─"──────────┤
                         │     └─{[original_char  orginal_char]}─┘                      │
                         ├─file:directory/filename──────────────────────────────────────┤
                         └─table:table.column────────────────────────────────────────────┘
```

Element	Description
column	The name of the column that contains canonical mapping strings.
directory	The directory path for the canonical mapping file.
filename	The name of the file that contains canonical mapping strings.

Element	Description
table	The name of the table that contains the column with canonical mapping strings.
original_char	The characters to replace with a mapped string during indexing and searching.
mapped_string	The characters that replace the original characters during indexing. If the *mapped_string* is empty, the original characters are not indexed.

Usage

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

You can update your canonical map by re-creating the **bts** index.

During indexing and searching, by default all characters are transformed to lowercase, 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 mapped characters are indexed and searched, therefore, when the results are returned, 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 that contain either Raùl or Raul are returned.

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

Example: Map characters as inline comma-separated strings

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

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

Example: 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 that have diacritical marks as uppercase. Therefore, both uppercase and lowercase versions of letters are included in the original characters.

```
{Ææ}:{ae},

{Œœ}:{oe},

{Ññ}:{ny},

{[ÀÁÂÃÅàáâââãå]}:{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 that specifies 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;
```

Example: Map single characters

The following example illustrates how to map a single character to another character. The original character is in the first set of braces and the character to map it to is in the second set of braces.

The following example maps the single character ù to the single character u:

```
{ù}:{u}
```

Example: Specify multiple original characters

The following examples illustrate how to specify multiple original characters. You can put multiple characters in a set of brackets and enclose the brackets in braces. Do not put a blank space between the characters when you use brackets or every blank space in the text is 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}
```

Example: Specify multiple characters in mapping strings

The following example illustrates how a mapping string can have multiple characters. For example, the following example maps the single æ character to the two letters ae:

```
{æ}:{ae}
```

Example: Prevent the indexing of characters

The following example prevents the indexing of the characters 's by specifying empty braces for the mapping string:

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

Example: Manage multiple spellings

The following example illustrates how to manage multiple spellings by mapping the possible strings to each other. 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 is:

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

Alternatively, you can map the two prefixes:

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

delete index parameter

The **delete** index parameter controls the optimizing, or compacting, of the index. Optimizing the index removes index information for deleted documents and releases disk space. You can optimize the **bts** index manually, which is the default mode, or automatically.

Optimize the index manually

When you create a **bts** index, the default mode for deleting rows is deferred (**delete="deferred"**). A delete operation on a row in a table marks the row as deleted in the **bts** index. The disk space can be reclaimed as more documents are added to the index. Queries that are run against **bts** columns do not return the deleted documents.

To release disk space that is occupied by the deleted documents in the index, use the **oncheck** utility in the format:

```
oncheck -ci -y db_name:table_name#index_name
```

Alternatively, you can use the **bts_index_compact()** function to release disk space for the rows marked for deletion. The difference between the two methods is that the **bts_index_compact()** function requires that you know the directory path to the **bts** index, whereas the **oncheck** utility requires that you know the database name, table name, and the index name. Both methods have the same result.

Delete operations are faster in the deferred mode. The deferred deletion mode is best for large indexes that are updated frequently.

Optimize the index automatically

You can override the deferred deletion mode by creating the **bts** index with the **delete="immediate"** parameter. In the immediate deletion mode, index information for deleted documents is physically removed from the index after every delete operation. This mode frees up space in the index immediately. However, the immediate deletion mode rewrites the index each time an index entry is deleted, which slows down delete operations and makes the index unusable during the delete operation.

Related reference:

Chapter 21, “Basic text search performance,” on page 21-1

oncheck -ci and -cI: Check index node links (Administrator's Reference)

field_token_max index parameter

If you have large documents, you can increase the maximum number of tokens that are indexed by setting the **field_token_max** index parameter to a positive integer up to 2 000 000 000. By default, 10 000 tokens are indexed in a document. If the average word has 5 characters, approximately 50-60 KB of the document is indexed.

Example

For example, the following statement creates a **bts** index that creates up to 500 000 tokens per document:

```
CREATE INDEX books_bts ON books(book_data bts_lvarchar_ops)
USING bts(field_token_max="500000") IN bts_sbspace;
```

max_clause_count index parameter

You can increase the maximum number of query results by setting the **max_clause_count** index parameter to a value greater than the default value of 1024.

Basic text queries fail when the maximum number of results is exceeded. If a query results in more than the maximum number of results, you receive the following error:

```
(BTSB0) - bts clucene error: Too Many Clauses
```

This error can occur during a wildcard or fuzzy search.

The limit of results controls virtual memory usage. Queries with large result sets can result in slower performance and the allocation of more virtual segments. You can monitor the number of virtual segments with the **onstat -g seg** command.

Example

The following statement creates a **bts** index with a maximum number of 4000 query results:

```
CREATE INDEX bts_idx ON bts_tab(text bts_char_ops)
USING bts (max_clause_count="4000")
IN sbspace1;
```

Related concepts:

“Wildcard searches” on page 16-5

“Fuzzy searches” on page 16-6

Related reference:

onstat -g seg command: Print shared memory segment statistics (Administrator's Reference)

query_default_field index parameter

Set the **query_default_field** index parameter to a column name to override the default field for basic text search queries from the `contents` field. Set the **query_default_field** index parameter to * to separately index each column in a composite index.

You do not need to specify the default field in a basic text search query. If you have a structured index on JSON or XML data, you can change the default field to one of the indexed field names or tags.

You can create a composite **bts** index on multiple text columns. By default, columns are concatenated and indexed as a single string in the `contents` field. Regardless of which column name you specify in the query, the matching text from all the indexed columns is returned. You can use the **query_default_field="*"** index parameter to index each column separately so that you can query by column name, which becomes the index field name. When you use the **query_default_field="*"** index parameter, only the matching text from the column name that you specify in the query is returned. You query multiple columns by including their field names in the format *fieldname*:*string*.

You cannot create a composite index on JSON or BSON columns.

If you combine the **query_default_field="*"** index parameter with the **xmltags** index parameter, the composite index is created on only the XML columns.

Examples: Create composite indexes

The following examples use a table with the following structure:

```
CREATE TABLE address(
    fname      char(32),
    lname      char(32),
    address1   varchar(64),
    address2   varchar(64),
    city       char(32),
    province   char(32),
    country    char(32),
    postalcode char(10)
);
```

You can create a composite **bts** index on multiple columns in the **address** table by using the following statement, which matches each column data type with its corresponding operator class:

```
CREATE INDEX  bts_idx ON address(
    fname      bts_char_ops,
    lname      bts_char_ops,
    address1   bts_varchar_ops,
    address2   bts_varchar_ops,
    city       bts_char_ops,
    province   bts_char_ops,
    country    bts_char_ops,
    postalcode bts_char_ops) USING bts;
```

The resulting composite index concatenates all the columns into the `contents` field. The following two queries would produce the same results because the text is not indexed by column name:

```
SELECT * FROM address WHERE bts_contains(fname, 'john');
SELECT * FROM address WHERE bts_contains(address1, 'john');
```

Alternatively, you can create a composite **bts** index and specify that each column is indexed separately by including the **query_default_field="*"** index parameter:

```
CREATE INDEX  bts_idx ON address(
    fname      bts_char_ops,
    lname      bts_char_ops,
    address1   bts_varchar_ops,
    address2   bts_varchar_ops,
    city       bts_char_ops,
    province   bts_char_ops,
    country    bts_char_ops,
    postalcode bts_char_ops) USING bts (query_default_field="*");
```

The resulting composite index includes the column name with the indexed text. The following two queries would produce different results:

```
SELECT * FROM address WHERE bts_contains(fname, 'john');
SELECT * FROM address WHERE bts_contains(address1, 'john');
```

The first query finds matches for `john` in the **fname** column and the second query finds matches for `john` in the **address1** column.

The following example searches for a row that contains specific text in two of its columns:

```
SELECT * FROM address WHERE bts_contains(fname, 'john AND city:nipigon');
```

This query returns the rows that contain both `john` in the **fname** column and `nipigon` in the **city** column.

Related reference:

Chapter 21, “Basic text search performance,” on page 21-1
“Basic Text Search query syntax” on page 16-1
“Basic text search index fields” on page 16-3

query_log index parameter

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

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

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

The fields are described in the following table.

Table 15-2. 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 15-2. Query tracking fields (continued)

Field name	Data type	Description
number of rows	INTEGER	The number of rows that are returned by the query.

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

This example shows how to track queries.

1. Create the **bts** index with tracking enabled:
```
CREATE INDEX bts_idx ON products (brands  bts_char_ops)
 USING bts (query_log="yes") IN sbsp1;
```
2. Create a table to hold the log records:
```
CREATE TABLE bts_query_log_data(
  qwhen DATETIME YEAR TO FRACTION,
  idx_name LVARCHAR,
  partn INTEGER,
  query LVARCHAR,
  rows INTEGER);
```
3. Load the log records into the log table:
```
LOAD FROM '$INFORMIXDIR/tmp/bts_query.log' INSERT INTO bts_query_log_data;
```
4. Query the log 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 austen
rows 3

3 row(s) retrieved.
```

stopwords index parameter

When you specify a customized stopword list, it replaces the default stopword list. You create a customized stopword list with the **stopwords** index parameter when you create the **bts** index.

stopwords index parameter:

```
                         ┌─,───────────────────────┐
                         ▼                         │
├──stopwords──=──"──┬─(────┬──────────┬──┬──────┬──┴─)─┬──"──────────────────┤
                    │      └─field──:─┘  └─word─┘      │
                    ├─file:directory/filename──────────┤
                    └─table:table.column───────────────┘
```

Element	Description
column	The name of the column that contains stopwords.
directory	The path for the stopwords file.

Element	Description
field	The XML tag, path, or the column name that is indexed.
filename	The name of the file that contains stopwords.
table	The name of the table that contains stopwords.
word	The term to use as a stopword. Stopwords must be lowercase.

Usage

You can create a stopword list for all fields or customized stopword lists for specific fields. Any words that are listed before any field names become the default stopword list, which is used for all fields not explicitly listed. All words that are listed after a field name and before the next field name are stopwords for the preceding field only. If a field is listed without any words following it, that field does not have a stopword list.

You can specify the list of stopwords in a table column or in a file. The file or table must be readable by the user who is creating the index. Separate the field name and stopword pairs in the file or table by commas, white spaces, new lines, or a combination of those separators. The file or table becomes read-only when the index is created. If you want to add or change stopword assignments, you must drop and re-create the index.

Examples

Example 1: Input stopwords as inline comma-separated words

Inline comma-separated words are useful when you have only a few stopwords. The following example prevents searching the words "am,", "be," and "are":

```
stopwords="(am,be,are)"
```

The following example shows how to create a **bts** index with an inline comma-separated customized stopword list:

```
CREATE INDEX books_bts ON books(book_data bts_lvarchar_ops)
USING bts(stopwords="(am,be,are)") IN bts_sbspace;
```

Example 2: Input stopwords from a file or a table column

The following example shows the contents of a stopword file where stopwords are separated by commas, white spaces, and new lines:

```
avec, et
mais pour
```

The following example shows how to create a **bts** index with a customized stopword list in a file:

```
CREATE INDEX books_bts ON books(book_data bts_lvarchar_ops)
USING bts(stopwords="file:/docs/stopwords.txt") IN bts_sbspace;
```

The following example shows how to create a **bts** index with a customized stopword list in a table column:

```
CREATE INDEX books_bts ON books(book_data bts_lvarchar_ops)
USING bts(stopwords="table:mytable.mycolumn") IN bts_sbspace;
```

Example 3: Create stopword lists for specific fields

The following example creates a stopword list of `am`, `be`, and `are` for all fields except the fields `author` and `title`, which have their own stopwords, and the field `edition`, which does not have any stopwords.

```
CREATE INDEX books_bts ON books(book_data bts_lvarchar_ops)
USING bts(stopwords=
    "(am,be,are,
      author:mrs,mr,ms,
      title:the,an,a,or,
      edition:)"
       )
IN bts_sbspace;
```

thesaurus index parameters

You can create a thesaurus so that basic text searches return synonyms as well as exact matches of specified words. A thesaurus is useful if your text data has multiple words for the same information.

thesaurus index parameters:

```
├──┬─thesaurus──=──"──yes──"──────────────────┬──────────────────────────────┤
   └─thesaurus_index──=──"──thesaurus_index──"─┘
```

Element	Description
thesaurus_index	The name of the **bts** index that is created on the thesaurus table.

Usage

People's names is an example of the type of data that can benefit from a thesaurus. Because people can have nicknames, multiple names for the same person might exist in the database. If you define a thesaurus for common nicknames, your basic text queries can return more accurate results. Synonyms are not used in a query if the query includes the following term modifiers: wildcard, fuzzy, proximity, or range query.

When you include a thesaurus in your **bts** index definition, basic text queries include all synonyms for specific search terms. For example, if you define mark, marc, marcus, and marco as synonyms, when you query for any one of these names the query is rewritten to include all of them:

```
'(mark OR marc OR marcus OR marco)'
```

To create a thesaurus:

1. Create the thesaurus table with a text column for the synonym data. You can use any of the data types that are supported by the **bts** index.
2. Add the synonym data to the thesaurus table. Each value for the synonym data column is a list of words that you want to be treated as synonyms. You can create synonyms for only single words. You cannot create synonyms for phrases.
3. Create a **bts** index on the thesaurus table. Include the **thesaurus="yes"** parameter.

When you create the **bts** index on the table that contains the text data, follow these rules:

- Specify the synonym data column as the column to index.
- Include the **thesaurus_index="***thesaurus_index***"** parameter, specifying the thesaurus index that you created.
- Set the **query_default_operator** index parameter to **"OR"** or omit the parameter.

You can dynamically update your thesaurus without rebuilding the basic text search index by updating the thesaurus table.

Example

Suppose that you create a table called **mytbl** with the following statements:

```
CREATE TABLE mytbl(name char(30));
INSERT INTO mytbl(name) VALUES('mark');
INSERT INTO mytbl(name) VALUES('elizabeth');
INSERT INTO mytbl(name) VALUES('marco');
INSERT INTO mytbl(name) VALUES('beth');
```

You create a thesaurus table named **mythesaurus** and add synonym data to it:

```
CREATE TABLE mythesaurus(synonyms lvarchar);
INSERT INTO mythesaurus(synonyms)
      VALUES('elizabeth liz beth eliza leisal betty liza');
INSERT INTO mythesaurus(synonyms)
      VALUES('mark marc marcus marco');
```

You create a **bts** index on the thesaurus table:

```
CREATE INDEX mythesaurus_index
ON mythesaurus(synonyms bts_lvarchar_ops)
USING bts(thesaurus="yes");
```

You create a **bts** index that uses the thesaurus on the table **mytbl**:

```
CREATE INDEX name_index
ON mytbl(name bts_char_ops)
USING bts(thesaurus_index="mythesaurus_index");
```

Now when you search for the name `elizabeth`, the query returns both the exact match and the synonym `beth`:

```
SELECT * FROM mytbl WHERE bts_contains(name, 'elizabeth');

name
elizabeth
beth
2 row(s) retrieved.
```

When you search for both `marcus` or `liza`, the query returns four synonyms but no exact matches:

```
SELECT * FROM mytbl WHERE bts_contains(name, 'marcus or liza');

name
mark
marco
elizabeth
beth
4 row(s) retrieved.
```

xact_memory index parameter

You can set a limit on the amount of memory that is used by basic text search operations. Restricting memory usage is useful if you have a low memory configuration. By default, the memory limit is set by the value of the SHMTOTAL configuration parameter.

xact_memory index parameter:

```
├──xact_memory──=──┬─memory_size─┬───┬─┬──────────────────────────────────────┤
                   │             ├─K─┤ │
                   │             ├─M─┤ │
                   │             └─G─┘ │
                   └─unlimited─────────┘
```

Element	Description
memory_size	A positive integer that represents the maximum amount of memory for **bts** index operations. The default unit is bytes. To specify a different multiple of bytes, include one of the following letters at the end of the number: • K = Kilobytes • M = Megabytes • G = Gigabytes For example, `5G` sets the maximum memory usage to 5 gigabytes.

Usage

If any **bts** index operation requires more memory than the value of the **xact_memory** index parameter, the operation fails.

If the **xact_memory** index parameter is set to **unlimited** or is not included in the index, the memory limit is set by the value of the SHMTOTAL configuration parameter.

Example

For example, the following statement creates a **bts** index that limits the amount of memory for basic text search transactions to 5 GB:

```
CREATE INDEX books_bts ON books(book_data bts_lvarchar_ops)
USING bts(xact_memory=5G) IN bts_sbspace;
```

Related reference:

SHMTOTAL configuration parameter (Administrator's Reference)

xact_ramdirectory index parameter

By default, you build a **bts** index in a temporary sbspace. You can build a **bts** index faster in RAM than in a temporary sbspace.

Include the **xact_ramdirectory="yes"** in the **bts** index to build the index in memory. However, when building the index in memory uses too much memory, the build is switched to the temporary sbspace. The maximum amount of memory that is allowed for an index build is approximately one third of the value of that is specified by the **xact_memory** index parameter or the SHMTOTAL configuration parameter, whichever is more restrictive.

Related reference:

Chapter 21, “Basic text search performance,” on page 21-1

Chapter 16. Basic text search queries

You perform basic text search queries with the **bts_contains()** search predicate.

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 “bts access method syntax” on page 15-5.

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

Use the **bts_contains()** search predicate to run basic text search queries.

You can also run a basic text search query with the Informix JSON **$ifxtext** query operator. Use the same syntax for the search criteria for both methods.

bts_contains() Search Predicate:

```
├──bts_contains──(──column──,──'──┤ Search criteria ├──'──┬──────────────────┬──────►
                                                          └─,──score # REAL──┘

►──)──────────────────────────────────────────────────────────────────────────┤
```

Search criteria:

```
├──┬──────────────────────────┬──query_string──────────────────────────────────┤
   │                     (1)  │
   └──┤ Index field ├──:──────┘
```

Notes:

1 See “Basic text search index fields” on page 16-3.

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

query_string
The search string. The search string includes the following elements:

Query term
Required. One or more words that you want to search for.

Query term modifiers
: Optional. You can modify query terms to run wildcard, fuzzy, proximity, and range searches. You can boost the importance of a query term relative to other terms.

Boolean operators
: Optional. You can include Boolean operators to combine query terms in logical combinations.

If an index has multiple fields because it is a structured or a composite index, you can include an index field name to modify the search string.

score # REAL
: Optional argument that is used to pass a statement local variable (SLV) to the text search engine. The search engine uses this variable to record the document score it assigns to each row in the results. The score value is a REAL number between 0.0 and 100.0 inclusive that indicates the relevance of each document to the search criteria, compared to that of other indexed records. The higher the document score value, the more closely the document matches the criteria.

The following example shows a search for the word `standard` in the column **brands** in a table called **products**.

```
SELECT id FROM products
WHERE bts_contains(brands, 'standard');
```

You can use an SLV as a filtering mechanism and to sort the results by score. The following example returns documents that contain the word `standard` from the column **brands** in a table that is called **products** if the document score value is greater than 70. The results are ordered in descending order by score.

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

Related concepts:

"query_default_field index parameter" on page 15-13

Related reference:

"XML index parameters syntax" on page 18-1

Informix query operators (JSON compatibility)

Basic Text Search query terms

Query terms are words or phrases.

A word is a single word, such as `Hello`. A phrase is a group of words that are enclosed in double quotation marks, such as `"Hello World"`. Multiple words or phrases can be combined with Boolean operators to form complex queries.

This example searches for the word `Coastal`:

```
bts_contains(column, 'Coastal')
```

This example searches for the phrase `"Black and Orange"`:

```
bts_contains(column, ' "Black and Orange" ')
```

White space and punctuation characters are ignored. Terms within angle brackets (< >) are not interpreted as tagged HTML or XML text unless you are using XML index parameters. Letter case is not considered in query terms. Words are indexed in lowercase according to the DB_LOCALE environment variable setting. All three of the following search predicate examples search for the term `orange8` in unstructured text:

```
bts_contains(column, ' Orange8 ')
bts_contains(column, ' <oranGe8> ')
bts_contains(column, ' "<Orange8>" ')
```

Grouping words and phrases

You can group words and phrases in parentheses to form more complex queries by including Boolean operators. For example, to search for words `UNIX` or `Windows` and the phrase `operating system`, you can use this search predicate:

```
bts_contains(column, ' (UNIX OR Windows) AND "operating system" ')
```

This search returns results that must contain the phrase `operating system`, and either `the word UNIX` or the word `Windows`.

You can also group words and phrases in field data:

```
bts_contains(column, ' os:(UNIX AND "Windows XP") ')
```

In that case, the search results must contain the word `UNIX` and the phrase `Windows XP` in the `os` field.

Escaping special characters

You can use the special characters that are part of basic text search query syntax in searches by using the backslash (\) as an escape character before the special character.

The following characters are Basic Text Search special characters: + - && || ! ( ) { } [ ] ^ " ~ * ? : \

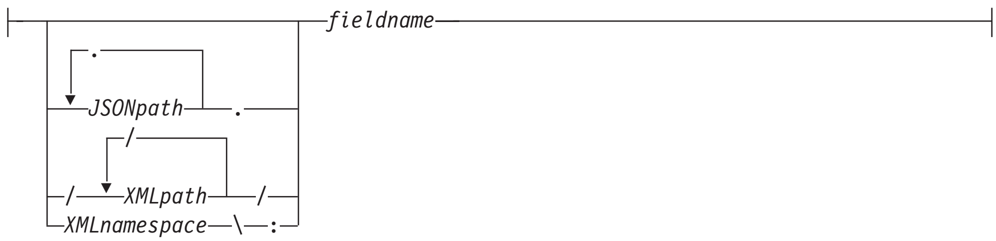
For example, to search for the phrase `(7+1)`, use the following search predicate:

```
bts_contains(column, ' \(7\+1\) ')
```

Basic text search index fields

The **bts** index indexes searchable data in *fields*. When you index unstructured text, each value is indexed in a default field called `contents`. You do not need to specify the default field in the **bts_contains()** search predicate. When you create an index that has multiple fields because it is a structured or a composite index, you might need to include a field name to modify the search string in the **bts_contains()** search predicate.

Index fields:



fieldname
: The name of the field that is indexed.

JSONpath
: If the **json_path_processing** index parameter is enabled, you can include the path before the field name. Separate each part of the path with a period.

XMLpath
: If the **xml_path_processing** index parameter is enabled, you can include the path before the field name. Separate each part of the path with a forward slash.

XMLnamespace
: If the **include_namespaces** index parameter is enabled, you can include an XML namespace before the field name. Escape the colon in the namespace with a back slash.

If you create a composite index on multiple columns, by default the text from the indexed columns is concatenated into one string and indexed in the `contents` field. To index the text in each column included in the index under a field of the same name, include the **query_default_field="*"** index parameter in the index definition. When you query on a composite index that has multiple fields, you must specify the field name in the **bts_contains()** search predicate.

Searches on structured JSON or XML indexes

When you index structured text by setting XML or JSON index parameters, the names for the XML tags or JSON field names are indexed in separate fields and you must specify those fields in the **bts_contains()** search predicate.

If you specify a list of XML tags or JSON field names to be indexed with the **xmltags** or **json_names** index parameter, the default field is the first field in the field list. You must specify the field name for any other field in the **bts_contains()** search predicate. However, you can override the default field by setting the **query_default_field** index parameter to a specific field name to use as the default field.

If you enable the **`all_xmltags`** or **all_json_names** index parameter, there is no default field. You must specify each field name in the **bts_contains()** search predicate.

Examples: JSON or BSON documents

For these examples, the following JSON document is indexed as field name-value pairs with paths by enabling the **all_json_names** and **json_path_processing** index parameters:

```
{ "person" : {
     "givenname" : "Jim"
}
```

For example, to search the `given name` field, you can use either of the following search predicates:

`bts_contains(`*`column`*`, ' givenname:Jim ')`

`bts_contains(`*`column`*`, ' givenname:"Jim" ')`

To search for a field that includes a path, include a period between the field name elements. For example, to search the `person:given name` field, you can use the following search predicate:

`bts_contains(`*`column`*`, ' person.givenname:"Jim" ')`

Examples: XML documents

For example, if the XML data is indexed in a field that is called `fruit`, you can use the following search predicates:

`bts_contains(`*`column`*`, ' fruit:Orange ')`

`bts_contains(`*`column`*`, ' fruit:"Orange Juice" ')`

If the XML data is indexed in a field that contains the path `/fruit/citrus`, you can use the following search predicate:

`bts_contains(`*`column`*`, ' /fruit/citrus:"Orange Juice" ')`

If you enable the **include_namespaces** index parameter, you must escape the colon (:) in namespaces with a backslash (\). For example, if you are using the `fruit` namespace:

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

Related concepts:

"query_default_field index parameter" on page 15-13

"The all_xmltags index parameter" on page 18-4

"The xmltags index parameter" on page 18-2

Related reference:

"bts_index_fields() function" on page 20-1

"all_json_names index parameter" on page 17-4

"json_names index parameter" on page 17-8

Basic Text Search query term modifiers

You can modify query terms to perform more complex searches.

If you are searching fielded data, you can use query term modifiers only on the query terms, not on the field names.

Wildcard searches

You can use wildcards in basic text search queries on single terms. You cannot use wildcards in searches on phrases.

To perform a single-character wildcard search, use a question mark (?) in the search term. The single-character wildcard search looks for terms that match with the single character replaced. For example, to search for the terms `text` and `test`, use `te?t` in the search predicate:

```
bts_contains(column, 'te?t')
```

You can use a single wildcard character (?) as the first character of the search term.

Multiple-character wildcard searches

Multiple-character wildcard searches look for zero or more characters.

To perform a multiple-character wildcard search, use an asterisk (*) in the search term. For example, to search for `geo`, `geography`, and `geology`, use `geo*` in the search predicate:

```
bts_contains(column, 'geo*')
```

The multiple-character wildcard search can also be in the middle of a term. For example, the search term `c*r` will match `contour`, `crater`, `color`, and any other words that start with the letter c and end with the letter r:

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

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

If the number of indexed tokens that match your wildcard query exceed 1024, you receive the following error:

```
(BTSB0) - bts clucene error: Too Many Clauses
```

To solve this problem, you can make the query more restrictive or you can recreate the **bts** index with the **max_clause_count** index parameter set to a number greater than 1024.

Related concepts:

"max_clause_count index parameter" on page 15-12

Fuzzy searches

A fuzzy search searches for text that matches a term closely instead of exactly. Fuzzy searches help you find relevant results even when the search terms are misspelled.

To perform a fuzzy search, append a tilde (~) at the end of the search term. For example the search term `bank~` will return rows that contain `tank`, `benk` or `banks`.

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

You can use an optional parameter after the tilde in a fuzzy search to specify the degree of similarity. The value can be between 0 and 1, with a value closer to 1 requiring the highest degree of similarity. The default degree of similarity is 0.5, which means that words with a degree of similarity greater than 0.5 are included in the search.

The degree of similarity between a search term and a word in the index is determined by using the following formula:

```
similarity = 1 - (edit_distance / min ( len(term), len(word) ) )
```

The edit distance between the search term and the indexed word is calculated by using the Levenshtein Distance, or Edit Distance algorithm. The **min()** function returns the minimum of the two values of the **len()** functions, which return the length of the search term and the indexed word. The following table shows the values used to calculate similarity and the resulting similarity between the search term "tone" and various indexed words.

Table 16-1. Sample set of comparisons

Term	Length of term	Word	Length of word	Edit distance	Similarity
tone	4	tone	4	0	1.00
tone	4	ton	3	1	0.67
tone	4	tune	4	1	0.75
tone	4	tones	4	1	0.75
tone	4	once	4	2	0.50
tone	4	tan	3	2	0.33
tone	4	two	3	3	0.00
tone	4	terrible	8	6	-0.50
tone	4	fundamental	11	9	-1.25

For example, the following query searches for words with the default degree of similarity of greater than 0.50 to the search term `tone`:

```
bts_contains(text, 'tone~')
```

This query returns rows that contain these words: `tone`, `ton`, `tune`, and `tones`. Rows that contain the word `once`are not included because the degree of similarity for `once` is exactly 0.50, not greater than 0.50. The following query would include the rows that contain the word `once`:

```
bts_contains(text, 'tone~0.49')
```

Tip: Test the behavior of specifying the degree of similarity with your data before you rely on it in your application.

If the number of indexed tokens that match your fuzzy query exceed 1024, you receive the following error:

```
(BTSB0) - bts clucene error: Too Many Clauses
```

To solve this problem, you can make the query more restrictive or you can recreate the **bts** index with the **max_clause_count** index parameter set to a number greater than 1024.

Related concepts:

"max_clause_count index parameter" on page 15-12

Proximity searches

You can specify the number of nonsearch words that can occur between search terms in a proximity search.

To perform a proximity search, enclose the search terms within double quotation marks and append a tilde (~) followed by the number of nonsearch words allowed. For example, to search for the terms `curb` and `lake` within 8 words of each other within a document, use the following search predicate:

```
bts_contains(column, ' "curb lake"~8 ')
```

Range searches

With a range search, you match terms that are between the lower and upper bounds specified by the query. Range searches can be inclusive or exclusive of the upper and lower bounds. Sorting is in lexicographical order (also known as dictionary order or alphabetic order).

Lexicographical order does not give the expected results to numeric data unless all numbers have the same number of digits. If necessary, add zeros to the beginning of numbers to provide the necessary number of digits.

Range searches use the keyword TO to separate search terms. By default, the word "to" is a stopword and is not an indexed term. If you are using a stopword list that does not include the word "to" or you are not using a stopword list, omit the word TO from the range query.

Inclusive range searches

Use brackets ([ ]) in the search predicate to specify an inclusive search. The syntax is [*searchterm1* TO *searchterm2*].

The following search predicate finds all terms between `apple` and `orange`, including the terms `apple` and `orange`:

```
bts_contains(column, ' [apple TO orange] ')
```

This example finds all terms between `20063105` and `20072401`, including `20063105` and `20072401`:

```
bts_contains(column, ' [20063105 TO 20072401] ')
```

Exclusive range searches

Use braces ({ }) in the search predicate to specify an exclusive search. The syntax is {*searchterm1* TO *searchterm2*}.

The following search predicate finds all terms between `Beethoven` and `Mozart`, excluding the terms `Beethoven` and `Mozart`:

```
bts_contains(column, ' {Beethoven TO Mozart} ')
```

This example finds all terms between `65` and `89`, excluding `65` and `89`:

```
bts_contains(column, ' {65 TO 89} ')
```

Boost a term

Boosting a term assigns more relevance to a word or phrase.

By default, all terms have equal value when the relevance score of a matching document is computed. Boosting a term raises the score of a document that contains it above the score of documents that do not. The search results are the same, but when sorted in descending order by score, documents containing the boosted term appear higher in the results.

To boost a term, use the caret symbol (^) followed by a number for the boost factor after the term that you want to appear more relevant. By default the boost factor is 1. It must be a positive number, but it can be less than one: for example .3 or .5.

For example, if your search terms are Windows and UNIX as in the search predicate bts_contains(*column*, ' Windows UNIX '), you can boost the term Windows by a factor of 4:

```
bts_contains(column, ' Windows^4 UNIX ')
```

This example boosts the phrase road bike over the phrase mountain bike by a factor of 2:

```
bts_contains(column, ' "road bike"^2 "mountain bike"  ')
```

You can also boost more than one term in a query. This example would return rows with the term lake before documents with the term land, before documents with the term air.

```
bts_contains(column, ' lake^20  land^10 air  ')
```

Tip: Test the behavior of boosting a term with your data before you rely on it in your application.

Boolean operators

Boolean operators combine terms in logical combinations. You can use the operators AND, OR, and NOT, or their equivalent special characters, in the **bts_contains()** search predicate.

By default, the OR operator is assumed if you do not supply a Boolean operator between two terms. However, you change the default operator to AND by setting the **query_default_operator** to AND when you create a **bts** index. For more information, see "bts access method syntax" on page 15-5.

The Boolean operators are not case-sensitive.

AND operator

The AND operator matches documents where both terms exist anywhere in the text of a single document.

You can also use two adjacent ampersands (&&) instead of AND.

If the **query_default_operator** index parameter is set to AND, the AND operator is assumed if you do not specify a Boolean operator between two terms.

The following search predicates search for documents that contain both the word UNIX and the phrase operating system:

```
bts_contains(column, ' UNIX AND "operating system"  ')
bts_contains(column, ' UNIX && "operating system"  ')
```

The following search predicates search XML data for documents that contain both the word travel in the book field and the word stewart in the author field:

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

The following search predicate searches for documents that contain both the word travel in the book field and the phrase john stewart in the author field:

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

OR operator

The OR Boolean operator is the default conjunction operator. If no Boolean operator appears between two terms, the OR operator is assumed, unless the **query_default_operator** index parameter is set to AND. In that case, you must specify the OR operator, or use two adjacent vertical bars (||) to represent the OR operator.

The following search predicates find documents that contain either the term `UNIX` or the term `Windows`:

```
bts_contains(column, ' UNIX Windows ')
bts_contains(column, ' UNIX OR Windows ')
bts_contains(column, ' UNIX || Windows ')
```

NOT operator

Use the NOT Boolean operator in combination with the AND operator (or its equivalent symbols) when you want to search for documents that do not contain a specified term or phrase.

The NOT operator can also be denoted with an exclamation point (!) or with a minus sign (-).

The following search predicates find documents that contain the term `UNIX`, but not the term `Windows`:

```
bts_contains(column, ' UNIX AND NOT Windows ')
bts_contains(column, ' UNIX AND !Windows ')
bts_contains(column, ' +UNIX -Windows ')
```

The minus sign (-) can be used with the plus sign (+), but not with the AND operator.

Chapter 17. Basic text search JSON index parameters

You can include JSON index parameters when you create a **bts** index to control how JSON and BSON columns are indexed.

By default, all field names and values are indexed as unstructured text in the `contents` field. Use JSON index parameters to control the following aspects of the **bts** index:

- Whether to index the documents as field name-value pairs so that you can search for text by field. Enable the **all_json_names** index parameter to index all field names. Set the **json_names** index parameters to index specific field names. You have the following choices to further refine how field name-value pairs are indexed:
  - Whether to include JSON or BSON object paths in field name-value pairs so that you can search based on the field hierarchy in the document. Enable the **json_path_processing** index parameter to index paths.
  - Whether to index the position of values in arrays so that you can search specific positions in arrays. Enable the **json_array_processing** index parameter to index the position of arrays.
  - Whether to index as both field name-value pairs and unstructured text so that you have the flexibility to search a specific field or all fields. Enable the **include_contents** index parameter to include an unstructured index of field names and values.
- Whether an unstructured index contains only values and no field names so that you do not receive field names in search results. Enable the **only_json_values** index parameter to limit the unstructured index to values.
- Whether to ignore format errors for JSON or BSON documents. Enable the **ignore_json_format_errors** index parameter to ignore incorrectly formatted documents.

Requirements and restrictions

The JSON or BSON documents must be in a UTF-8 locale.

Any XML values in a JSON or BSON document are indexed as unstructured text.

The following parts of JSON or BSON documents are indexed by a **bts** index:

- JSON string values, or the corresponding BSON element code 0x2.
- JSON number values, which are converted to string representations: 4-byte integers, 8-byte integers, and 8-byte floating points, or the corresponding BSON element codes: \x01, \x09, \x10, \x11, and \x12.
- JSON TimeStamp and Coordinated Universal Time Datetime values, which are converted to string representations

The following parts of JSON or BSON documents are not indexed:

- JSON Boolean true, Boolean false, and null values
- The BSON element codes: 0x05, 0x06, 0x07, 0x08, 0x0A, 0x0B, 0x0C, 0x0D, 0x0E, 0x0F, 0xFF, and 0x7F.
- Any name-value pair that has a zero-length field name
- Fields that contain numbers

You cannot create a composite index on a JSON or BSON column.

Example document

The examples for indexing JSON and BSON documents are based on the following JSON document, which is assumed to be in the **docs** column of the **json_tab** table:

```
{ "person" : {
     "givenname" : "Jim",
     "surname" : "Flynn",
     "age" : 29,
     "cars" : [ "dodge", "olds" ],
     "parents":[
        { "givenname" : "Slim",
          "surname" : "Flynn" },
        { "givenname" : "Lynn",
          "surname" : "Kim" }
     ]
}
```

The **bts** index on a JSON or BSON document is based on a tree representation of the document. You need to understand the tree representation if you include paths or array positions in the field name-value pairs of a structured index. The example JSON document has the following tree representation:

```
 "person".
          "givenname" : "Jim"
          "surname" : "Flynn"
          "age" : "29",
          "cars".
                 "0" : "dodge"
                 "1" : "olds"
          "parents".
                    "0".
                        "givenname" : "Slim"
                        "surname" : "Flynn"
                    "1".
                        "givenname" : "Lynn"
                        "surname" : "Kim"
```

JSON index parameters syntax

You can use JSON index parameters to index the contents of JSON and BSON columns as structured or unstructured text, or both.

Include JSON index parameters in the **bts** index definition when you create the **bts** index. See “bts access method syntax” on page 15-5. You can also create a **bts** index on a BSON column by running the Informix JSON **createTextIndex** command. Both methods requires the same syntax for JSON and other **bts** index parameters.

You can index JSON or BSON documents as structured or unstructured text.

JSON index parameters for structured text:

```
                                                  (1)
├──┬──┤ The json_names index parameter ├──┬──────────────────────────────────────────►
   │                                (2)  │
   └─all_json_names=──┬─"yes"─┬──────────┘
                      └─"no"──┘
```

```
►─┬────────────────────────────────────────────────────────────────────────┬──────────►
  │                        (3)                                             │
  └─,include_contents=─┬─"yes"─┬─────────────────────────────────────────┬─┘
                       └─"no"──┘ │                                    (4)│
                                 └─,only_json_values=─┬─"yes"─┬──────────┘
                                                      └─"no"──┘

►─┬─────────────────────────────────────┬─┬────────────────────────────────────────────┬──►
  │                            (5)      │ │                                       (6)  │
  └─,json_path_processing=─┬─"yes"─┬────┘ └─┬───┬─json_array_processing=─┬─"yes"─┬────┘
                           └─"no"──┘        └─,─┘                        └─"no"──┘

►─┬──────────────────────────────────────────────┬─────────────────────────────────────►◄
  │                                   (7)        │
  └─┬───┬─ignore_json_format_errors=─┬─"yes"─┬───┘
    └─,─┘                            └─"no"──┘
```

Notes:

1 See "json_names index parameter" on page 17-8.

2 See "all_json_names index parameter" on page 17-4.

3 See "include_contents index parameter" on page 17-5.

4 See "only_json_values index parameter" on page 17-11.

5 See "json_path_processing index parameter" on page 17-10.

6 See "json_array_processing index parameter" on page 17-6.

7 See "ignore_json_format_errors index parameter" on page 17-5.

JSON index parameters for unstructured text:

```
├─┬────────────────────────────────────┬─┬────────────────────────────────────────────┬──►
  │                          (1)       │ │                                       (2)  │
  └─only_json_values=─┬─"yes"─┬────────┘ └─┬───┬─json_array_processing=─┬─"yes"─┬────┘
                      └─"no"──┘             └─,─┘                        └─"no"──┘

►─┬──────────────────────────────────────────────┬─────────────────────────────────────►◄
  │                                   (3)        │
  └─┬───┬─ignore_json_format_errors=─┬─"yes"─┬───┘
    └─,─┘                            └─"no"──┘
```

Notes:

1 See "only_json_values index parameter" on page 17-11.

2 See "json_array_processing index parameter" on page 17-6.

3 See "ignore_json_format_errors index parameter" on page 17-5.

Usage

If you do not include any JSON index parameters when you create a **bts** index on a JSON or BSON column, both the field names and the values are indexed together as unstructured text.

Include a comma between parameters.

Example

The following statement creates a **bts** index without JSON index parameters on the example JSON **docs** column:

```
CREATE INDEX bts_idx
        ON json_tab (docs bts_json_ops)
        USING bts;
```

The resulting index contains the following unstructured text that is based on the tree representation of the document in the `contents` field:

```
contents: person givenname jim surname flynn age 29 cars dodge olds parents
givenname slim surname flynn givenname lynn surname kim
```

Related reference:

Informix JSON commands (JSON compatibility)

all_json_names index parameter

Enable the **all_json_names** index parameter to index JSON or BSON documents as field name-value pairs instead of as unstructured text.

All the field names in the documents in the column are indexed as fields in the **bts** index. When you query on the JSON or BSON column, you must specify the field name to search in the **bts_contains()** search predicate.

You can include the **json_path_processing** and **json_array_processing** index parameters to add the paths and array positions to the field names.

To view the fields that you indexed, run the **bts_index_fields()** function.

Example: Index all field name-value pairs

The following statement creates a **bts** index with the **all_json_names** index parameter enabled on the example JSON **docs** column:

```
CREATE INDEX bts_idx
         ON json_tab (docs bts_json_ops)
       USING bts(all_json_names="yes");
```

The resulting index contains the following field name-value pairs:

```
givenname: jim
surname: flynn
givenname: slim
surname: flynn
age: 29
cars: dodge
cars: olds
givenname: lynn
surname: kim
```

You can specify the following fields in the search predicate: **givenname**, **surname**, **age**, and **cars**.

Related reference:

“bts_index_fields() function” on page 20-1
“Basic text search index fields” on page 16-3

ignore_json_format_errors index parameter

Enable the **ignore_json_format_errors** index parameter to skip inserting incorrectly formatted JSON or BSON documents and continue processing the SQL statement. By default, if you attempt to insert a JSON or BSON document that contains a format error, such as a missing brace or bracket, the entire SQL statement fails.

When you create a **bts** index or have an existing **bts** index and you insert JSON or BSON documents, the database server checks the formatting of the documents. When you enable the **ignore_json_format_errors** index parameter, incorrectly formatted documents are not inserted, but the rest of the statement continues processing. Any skipped documents result in messages in the online message log.

Example

The following statement creates a **bts** index with the **ignore_json_format_errors** index parameter enabled on the example JSON **docs** column:

```
create index bts_idx
         on json_tab (docs bts_json_ops)
      using bts(ignore_json_format_errors="yes");
```

Related reference:

Chapter 22, “Basic text search error codes,” on page 22-1

include_contents index parameter

Enable the **include_contents** index parameter to index JSON or BSON documents as unstructured text as well as indexing the documents as field name-value pairs.

You can enable the **include_contents** index parameter if the **all_json_names** parameter is enabled or the **json_names** parameter is specified.

By default, both field names and values are indexed in the `contents` field. If you enable the **only_json_values** index parameter, only the values are indexed in the `contents` field.

Example: Index all fields as field name-value pairs and unstructured text

The following statement creates a **bts** index with the **all_json_names** and **include_contents** index parameters enabled on the example JSON **docs** column:

```
CREATE INDEX bts_idx
         ON json_tab (docs bts_json_ops)
      USING bts(
                all_json_names="yes",
                include_contents="yes");
```

The resulting index contains the following 9 field name-value pairs, and the field names and values as unstructured text in the `contents` field:

```
givenname: jim
givenname: slim
givenname: lynn
age: 29
cars: dodge
cars: olds
surname: flynn
surname: flynn
```

```
surname: kim
contents: person givenname jim surname flynn age 29 cars dodge olds
parents givenname slim surname flynn givenname lynn surname kim
```

Example: Index specified field name-value pairs, paths, and values as unstructured text

The following statement creates a **bts** index with the **json_names**, **json_path_processing**, **only_json_values**, and **include_contents** index parameters enabled on the example JSON **docs** column:

```
create index bts_idx
        on json_tab (docs bts_json_ops)
      using bts(
              json_names="(person.givenname,parents.surname)",
              json_path_processing="yes",
              include_contents="yes",
              only_json_values="yes");
```

The resulting index contains four fields: three field name-value pairs with paths and the unstructured text in the `contents` field:

```
person.givenname: jim
parents.surname: flynn
parents.surname: kim
contents: jim flynn 29 dodge olds slim flynn lynn kim
```

json_array_processing index parameter

Enable the **json_array_processing** index parameter to index the array positions of values in JSON or BSON documents as field names.

Array positions are numbers, starting with 0, which represent the position of the value in the array. For example, the array `"cars" : [ "dodge", "olds" ]` has two positions:

```
"cars".
        "0" : "dodge"
        "1" : "olds"
```

In this example, the field name for `dodge` is `0` and the field name for `olds` is `1`. Field names that are only numbers cannot be queried, and are therefore not indexed. If you index field name-value pairs and array positions, but not paths, then field name-value pairs in arrays are not indexed, because the field names are numbers.

Indexing the array positions is most useful when you also index field name-value pairs and paths. Array positions in a field name that includes a path are indexed because the field name contains more than just a number. For example, the field names with paths from the example array are `cars.0` and `cars.1`.

When array and path processing are both enabled, the paths specified in the **json_names** index parameter must include array positions.

Example: Index array positions

The following statement creates a **bts** index with the **json_array_processing** index parameter enabled on the example JSON **docs** column:

```
CREATE INDEX bts_idx
         ON json_tab (docs bts_json_ops)
       USING bts(
                json_array_processing="yes");
```

The resulting index indexes the following unstructured text that contains the values and the array positions in the document in the `contents` field:

```
contents: person givenname jim surname flynn age 29 cars 0 dodge 1 olds
parents 0 givenname slim surname flynn 1 givenname lynn surname kim
```

In this example, indexing the array positions does not provide meaningful index entries because the position numbers are not differentiated from other values. If you query for the number 1, you do not know if the number is a value or an array position. Array positions are meaningful only in the context of field names and paths.

Example: Index all field name-value pairs, paths, and array positions

The following statement creates a **bts** index with the **all_json_names**, **json_path_processing**, and **json_array_processing** index parameters enabled on the example JSON **docs** column:

```
CREATE INDEX bts_idx
         ON json_tab (docs bts_json_ops)
       USING bts(
                all_json_names="yes",
                json_path_processing="yes",
                json_array_processing="yes");
```

The resulting index contains the following field name-value pairs that contain paths and array positions:

```
person.givenname: jim
person.surname: flynn
person.age: 29
person.cars.0: dodge
person.cars.1: olds
person.parents.0.givenname: slim
person.parents.0.surname: flynn
person.parents.1.givenname: lynn
person.parents.1.surname: kim
```

Example: Index specified field name-value pairs, paths, and array positions

The following statement creates a **bts** index with the **json_names**, **json_path_processing**, and **json_array_processing** index parameters enabled on the example JSON **docs** column:

```
CREATE INDEX bts_idx
         ON json_tab (docs bts_json_ops)
       USING bts(
                json_names="(person.givenname,parents.1.surname)",
                json_path_processing="yes",
                json_array_processing="yes");
```

The array position is required. If you specify `parents.surname` instead of `parents.1.surname`, this example results in an error.

The resulting index contains the following field name-value pairs that contain paths and array positions:

```
person.givenname: jim
parents.1.surname: kim
```

Example: Index all field name-value pairs and array positions

The following statement creates a **bts** index with the **all_json_names** and **json_array_processing** index parameters enabled on the example JSON **docs** column:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(
                all_json_names="yes",
                json_array_processing="yes");
```

The resulting index contains the following field name-value pairs and array positions:

```
givenname: jim
givenname: slim
givenname: lynn
age: 29
surname: flynn
surname: flynn
surname: kim
```

The following field name-value pairs are not indexed because the field names are numbers:

```
0: dodge
1: olds
```

json_names index parameter

Enable the indexing of specific field name-value pairs in JSON or BSON documents with the **json_names** index parameter.

The input for the field names for the **json_names** index parameter can be a comma-separated list of names, an external file, or a table column.

The json_names index parameter:

```
                            ┌─,───────┐
                            ▼         │
├──json_names──=──"────┬─(──┴─field───┴──)─┬─────"────────────────────────────────┤
                       ├─file:directory/filename─┤
                       └─table:table.column──────┘
```

Table 17-1. Elements for the json_names index parameter

Element	Description
column	The column that contains the field names to index. Separate field names by commas, white spaces, or new-line characters.
directory	The location of the file that contains field names to index.
field	The field name to index.

Table 17-1. Elements for the json_names index parameter (continued)

Element	Description
filename	The name of the file that contains field names to index. Must be readable by the user who creates the index. Separate field names by commas, white spaces, or new-line characters.
table	The name of the table with the column that contains the field names to index. Must be readable by the user who creates the index.

The field names that you specify are indexed as fields in the **bts** index. The values in the fields can be searched. When you query on the JSON or BSON column, you must specify the field name to search in the **bts_contains()** search predicate. In searches, the default field is the first tag or path in the field list. The **bts** index does not check whether the fields exist in the column, which means that you can specify fields that you will add to the column after you create the index.

If you enable the **json_path_processing** index parameter, the field name can include relative or full paths. If you enable the **json_array_processing** index parameter, the field name can include array positions.

If you want to add new field names to the index, you must drop the index, update the field name list, and then re-create the index.

To view the fields that you indexed, run the **bts_index_fields()** function.

Example: Index one field name-value pair

The following statement creates a **bts** index with the **json_names** index parameter set to a single field name on the example JSON **docs** column:

```
CREATE INDEX bts_idx
         ON json_tab (docs bts_json_ops)
      USING bts(json_names="surname");
```

The resulting index contains the following field name-value pairs:

```
surname: flynn
surname: flynn
surname: kim
```

You must specify the **surname** field in the search predicate.

Example: Index field name-value pairs from a file

The following statement creates a **bts** index with the **json_names** index parameter set to a file that is named `jsonfield.txt`:

```
CREATE INDEX bts_idx
         ON json_tab (docs bts_json_ops)
      USING bts(json_names="file:/jsonfield.txt");
```

Example: Index field name-value pairs from a column

The following statement creates a **bts** index with the **json_names** index parameter set to a column that is named **jsonnames** in the **json_ref** table:

```
CREATE INDEX bts_idx
         ON json_tab (docs bts_json_ops)
       USING bts(json_names="table:json_ref.jsonnames");
```

Related reference:

"bts_index_fields() function" on page 20-1

"Basic text search index fields" on page 16-3

json_path_processing index parameter

Enable the **json_path_processing** index parameter to include paths as part of the field names in field-value pairs from JSON or BSON documents.

You can enable the **json_path_processing** index parameter if you enable the indexing of field name-value pairs with either one of the following index parameters:

- The **json_names** index parameter: In the list of fields to index, you can specify relative paths or full paths. For example, if the full path is `person.parents.surname`, you can specify the relative path `parents.surname`.
- The **all_json_names** index parameter: Full paths are indexed for all fields.

If you add the **json_array_processing** index parameter, the paths include array positions, for example: `person.cars.0`.

Example: Index all field name-value pairs and paths

The following statement creates a **bts** index with the **all_json_names** and **json_path_processing** index parameters enabled on the example JSON **docs** column:

```
CREATE INDEX bts_idx
         ON json_tab (docs bts_json_ops)
       USING bts(
                all_json_names="yes",
                json_path_processing="yes");
```

The resulting index contains the following field name-value pairs that include paths:

```
person.givenname: jim
person.surname: flynn
person.age: 29
person.cars: dodge
person.cars: car
person.parents.givenname: slim
person.parents.surname: flynn
person.parents.givenname: lynn
person.parents.surname: kim
```

Example: Index specified field name-value pairs and paths

The following statement creates a **bts** index with the **json_names** and **json_path_processing** index parameters on the example JSON **docs** column:

```
CREATE INDEX bts_idx
         ON json_tab (docs bts_json_ops)
       USING bts(
                json_names="(parents.surname,
                             person.givenname)",
                json_path_processing="yes");
```

The `parents.surname` path is a relative path instead of the full path, `person.parents.surname`.

The resulting index contains the following field name-value pairs that include paths:

```
person.givenname: jim
parents.surname: flynn
parents.surname: kim
```

only_json_values index parameter

Enable the **only_json_values** index parameter to index only the values in the JSON or BSON documents as unstructured text. The field names are not indexed.

You can index both field name-value pairs and values, but not field names, as unstructured text. The **json_names** or **all_json_names** index parameter enables the indexing of field name-value pairs. The **include_contents** index parameter enables the indexing of field names and values as unstructured text. Add the **only_json_values** index parameter to modify the behavior of the **include_contents** index parameter to omit field names from the `contents` field.

Example: Index values as unstructured text

The following statement creates a **bts** index with the **only_json_values** index parameter enabled on the example JSON **docs** column:

```
create index bts_idx
         on json_tab (docs bts_json_ops)
       using bts(only_json_values="yes");
```

The resulting index indexes the following unstructured text that contains only the values in the document in the `contents` field:

```
contents: jim flynn 29 dodge olds slim flynn lynn
```

Example: Index all field name-value pairs and values as unstructured text

The following statement creates a **bts** index with the **all_json_names**, **only_json_values**, and **include_contents** index parameters enabled on the example JSON **docs** column:

```
create index bts_idx
         on json_tab (docs bts_json_ops)
       using bts(
                all_json_names="yes",
                include_contents="yes",
                only_json_values="yes");
```

The resulting index contains the following 9 field name-value pairs and the values as unstructured text in the `contents` field:

```
givenname: jim
givenname: slim
givenname: lynn
age: 29
cars: dodge
cars: olds
surname: flynn
surname: flynn
surname: kim
contents: jim flynn 29 dodge olds slim flynn lynn kim
```

Chapter 18. Basic Text Search XML index parameters

This chapter describes the XML index parameters for basic text search and provides detailed examples about each parameter's usage.

XML index parameters syntax

You can use XML index parameters to index XML tag and attribute values in separate fields either by tag name, attribute name, or by path.

When you do not use XML index parameters, XML documents are indexed as unstructured text. The XML tags, attributes, and values are included in searches and are indexed together in the `contents` field.

Any JSON or BSON documents in an XML document are indexed as unstructured text.

Include XML index parameters in the **bts** index definition when you create the **bts** index. See “bts access method syntax” on page 15-5.

XML Index Parameters:

```
                              ┌─,───┐
                              ▼     │                        (1)
├──┬─xmltags──=──"──┬─(───field───┴─)────────┬──"──────────┬──────────────►
   │                ├─file:directory/filename─┤            │
   │                └─table:table.column──────┘            │
   │ ┌─,────────────────────────────────────┐              │
   │ ▼                             (2)      │              │
   └───┬─all_xmltags──=──┬─"no"──┬──────────┴──────────────┘
       │                 └─"yes"─┘       │
       │                              (3)│
       └─all_xmlattrs──=──┬─"no"──┬──────┘
                          └─"yes"─┘

   ┌─,──────────────────────────────────────────┐
   ▼                                            │
►───┬─────────────────────────────────────────┬─┴──────────────────────────┤
    │                                 (4)     │
    ├─include_contents──=──┬─"no"──┬──────────┤
    │                      └─"yes"─┘          │
    │                                 (5)     │
    ├─xmlpath_processing──=──┬─"no"──┬────────┤
    │                        └─"yes"─┘        │
    │                             (6)         │
    ├─strip_xmltags──=──┬─"no"──┬─────────────┤
    │                   └─"yes"─┘             │
    │                                 (7)     │
    ├─include_namespaces──=──┬─"no"──┬────────┤
    │                        └─"yes"─┘        │
    │                                  (8)    │
    └─include_subtag_text──=──┬─"no"──┬───────┘
                              └─"yes"─┘
```


Notes:

1 See “The xmltags index parameter.”

2 See “The all_xmltags index parameter” on page 18-4.

3 See “The all_xmlattrs index parameter” on page 18-5.

4 See “The include_contents index parameter” on page 18-9.

5 See “The xmlpath_processing index parameter” on page 18-7.

6 See “The strip_xmltags index parameter” on page 18-9.

7 See “The include_namespaces index parameter” on page 18-10.

8 See “The include_subtag_text index parameter” on page 18-12.

Table 18-1. Options for XML index parameters

Element	Description
column	The column that contains tags to index.
directory	The location of the file that contains tags to index.
field	The XML tag or path to index. The field values can be full or relative XML paths if used with the **xmlpath_processing** parameter.
filename	The name of the file that contains tags to index.
table	The name of the table that contains the column with tags to index.

Example

For example, you have the following XML fragment:

```
<skipper>Captain Black</skipper>
```

You can create a **bts** index for searching the text within the `<skipper> </skipper>` tags:

```
CREATE INDEX boats_bts ON boats(xml_data bts_lvarchar_ops)
USING bts(xmltags="(skipper)") IN bts_sbspace;
```

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

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

Related reference:

“Basic Text Search query syntax” on page 16-1

The xmltags index parameter

Use the **xmltags** parameter to specify which XML tags or XML paths are searchable in a column.

The XML tags or paths that you specify become the field names in the **bts** index. The text values within fields can be searched. In searches, the default field is the first tag or path in the field list. The Basic Text Search module does not check if the tags exist in the column, which means that you can specify fields for tags that you will add to the column after you have created the index.

The input for the field names for the **xmltags** parameter can be one of three forms:

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

Input as inline comma-separated field names

Inline comma-separated field names are useful when you have only a few fields to index. For example, `xmltags="(`*field1*`,`*field2*`,`*field3*`)"` where *fieldn* specifies the tag or path to index.

If the **xmltags** parameter is enabled, you can specify paths for the **xmltags** values. For example

```
xmltags="(/text/book/title,/text/book/author,/text/book/date)"
```

XML tags are case-sensitive. When you use the inline comma-separated field names for input, the field names are transformed to lowercase characters. If the field names are uppercase or mixed case, use an external file or a table column for input instead.

Input from a file or a table column

Input from an external file has the format: `xmltags="file:/`*directory*`/`*filename*`"`

Input from a table column has the format: `xmltags="table:`*table*`.`*column*`"`

The file or table that contains the field names must be readable by the user creating the index. The file or table is read only when the index is created. If you want to add new field names to the index, you must drop and re-create the index. The field names in the file or table column can be separated by commas, white spaces, newlines, or a combination.

Following is an example of how field names can appear in the file or the table column:

```
title, author
date ISBN
```

If the **xmlpath_processing** parameter is enabled, you can specify paths or combination of paths and individual field names in the file or the table column:

```
/text/book/title
author
```

For information about using XML paths, see “The xmlpath_processing index parameter” on page 18-7.

If you want to index all the XML tags in a column, see “The all_xmltags index parameter” on page 18-4.

To view the fields that you have indexed, use the **bts_index_fields()** function. See “bts_index_fields() function” on page 20-1.

Related reference:

“bts_index_fields() function” on page 20-1

"Basic text search index fields" on page 16-3

Example: Index specific XML tags

You can use the **xmltags** parameter to index-specific fields so that you can restrict your searches by XML tag names.

Given the table:

```
EXECUTE PROCEDURE IFX_ALLOW_NEWLINE('t');

 CREATE TABLE boats(docid integer, xml_data lvarchar(4096));
 INSERT INTO boats values(1, '
  <boat>
   <skipper>Captain Jack</skipper>
   <boatname>Black Pearl</boatname>
  </boat> ');
 INSERT INTO boats values(2, '
  <boat>
   <skipper>Captain Black</skipper>
   <boatname>The Queen Anne's Revenge</boatname>
   </boat> ');
```

To create a **bts** index for the `skipper` and `boatname` tags:

```
CREATE INDEX boats_bts ON boats(xml_data bts_lvarchar_ops)
USING bts(xmltags="(skipper,boatname)") IN bts_sbspace;
```

The index will contain the following fields:

For the row where docid = 1, the fields are:

```
skipper:Captain Jack
boatname:Black Pearl
```

For the row where docid = 2, the fields are:

```
skipper:Captain Black
boatname:The Queen Anne's Revenge
```

To search for the skipper with the name " Black", the SELECT statement is:

```
SELECT xml_data FROM boats WHERE bts_contains(xml_data, 'skipper:black');
```

The search will return docid 2 because the `skipper` field for that row contains the word "black." For docid = 1, the `boatname` field also contains the word "black," but it is not returned because the search was only for the `skipper` field.

The all_xmltags index parameter

Use the **all_xmltags** parameter to enable searches on all the XML tags or paths in a column.

All the XML tags are indexed as fields in the **bts** index. If you use the **xmlpath_processing** parameter, full paths are indexed. The text value within fields can be searched. The attributes of XML tags are not indexed in a field unless you use the **all_xmlattrs** index parameter.

For information about using paths, see "The xmlpath_processing index parameter" on page 18-7.

If you want to index only specific tags in a column, use the **xmltags** parameter. See "The xmltags index parameter" on page 18-2.

To view the fields that you have indexed, use the **bts_index_fields()** function. See "bts_index_fields() function" on page 20-1.

Related reference:

"bts_index_fields() function" on page 20-1

"Basic text search index fields" on page 16-3

Example: Index all XML tags

You can use the **all_xmltags** parameter to index all of the tags in a column.

Given the XML fragment:

```
<book>
 <title>Graph Theory</title>
 <author>Stewart</author>
 <date edition="second">January 14, 2006</date>
</book>
```

To create an index for all the XML tags, use the SQL statement:

```
CREATE INDEX book_bts ON books(xml_data bts_lvarchar_ops)
USING bts(all_xmltags="yes") IN bts_sbspace;
```

The index will contain three fields that can be searched:

```
title:graph theory
author:stewart
date:january 14, 2006
```

The top level `<book></book>` tags are not indexed because they do not contain text values. The `edition` attribute is also not indexed.

If you enable path processing with the **`xmlpath_processing`** parameter, you can index the full paths:

```
CREATE INDEX book_bts ON books(xml_data bts_lvarchar_ops)
USING bts(all_xmltags="yes",xmlpath_processing="yes") IN bts_sbspace;
```

The index will contain three fields with full paths that can be searched:

```
/book/title:graph theory
/book/author:stewart
/book/date:january 14, 2006
```

The all_xmlattrs index parameter

Use the **`all_xmlattrs`** parameter to search on XML attributes in a document repository stored in a column of a table. This parameter enables searches on all attributes that are contained in the XML tags or paths in a column that contains an XML document.

Specify an attribute using the syntax `@attrname`, where `attrname` is the name of the attribute.

All the XML attributes are indexed as fields in the **bts** index. If you use the **`xmlpath_processing`** parameter, full paths are indexed. The text value within fields can be searched. The tags of XML tags are not indexed in a field unless you use the **`all_xmltags`** index parameter.

To view the fields that you have indexed, use the **bts_index_fields()** function. See "bts_index_fields() function" on page 20-1.

Examples: Index XML attributes

These examples are based on the following three rows of data:

```
<boat><name reg="hmc">titanic</name></boat>

<airplane callsign="qofz">kittyhawk</airplane>

<boat><name reg="CAN">Spirit of Canada</name></boat>
```

Example 1: Compare all_xmltags and all_xmlattrs

The following CREATE INDEX statement uses the **all_xmltags** parameter:

```
CREATE INDEX bts_idx ON bts_100_tab(col2 bts_nvarchar_ops)
       USING bts(all_xmltags="yes") IN bts_sbspace1 ;
```

The index has these fields representing the type of tag:

```
airplane
name
```

By contrast, the following CREATE INDEX statement uses the **all_xmlattrs** parameter instead of the **all_xmltags** parameter:

```
CREATE INDEX bts_idx ON bts_100_tab(col2 bts_nvarchar_ops)
       USING bts(all_xmlattrs="yes") IN bts_sbspace1 ;
```

The index has these fields representing the attributes of the tags:

```
@callsign
@reg
```

Example 2: Combine all_xmlattrs and all_xmltags

The following CREATE INDEX statement uses both the **all_xmlattrs** and the **all_xmltags** parameters:

```
CREATE INDEX bts_idx ON bts_100_tab(col2 bts_nvarchar_ops)
       USING bts(all_xmlattrs="yes",
                 all_xmltags="yes") IN bts_sbspace1 ;
```

The index has these fields representing both the types of tags and the tag attributes:

```
@callsign
@reg
airplane
name
```

Example 3: Combine all_xmlattrs, all_xmltags, and xmlpath_processing

The following CREATE INDEX statement uses the **all_xmlattrs**, the **all_xmltags**, and the **xmlpath_processing** parameters:

```
CREATE INDEX bts_idx ON bts_100_tab(col2 bts_nvarchar_ops)
        USING bts(xmlpath_processing="yes",
                  all_xmlattrs="yes",
                  all_xmltags="yes") IN bts_sbspace1 ;
```

The index has these fields, representing the full paths of the tags and attributes:

```
/airplane
/airplane@callsign
/boat/name
/boat/name@reg
```

Example 4: Comparing all_xmltags to all_xmlattrs along with xmlpath_processing

The following CREATE INDEX statement uses the **all_xmltags** parameter with the **xmlpath_processing** parameter:

```
CREATE INDEX bts_idx ON bts_100_tab(col2 bts_nvarchar_ops)
        USING bts(xmlpath_processing="yes",
                  all_xmltags="yes") IN bts_sbspace1 ;
```

The index has these fields, representing the paths of the tags:

```
/airplane
/boat/name
```

The following CREATE INDEX statement uses the **all_xmlattrs** parameter with the **xmlpath_processing** parameter:

```
CREATE INDEX bts_idx ON bts_100_tab(col2 bts_nvarchar_ops)
        USING bts(xmlpath_processing="yes",
                 all_xmlattrs="yes") IN bts_sbspace1 ;
```

The index has these fields, representing the paths of the attributes:

```
/airplane@callsign
/boat/name@reg
```

The xmlpath_processing index parameter

Use the **xmlpath_processing** parameter to enable searches based on XML paths.

The **xmlpath_processing** parameter requires that you specify tags with the **xmltags** parameter or that you enable the **all_xmltags** or **all_xlmattrs** parameter.

When you enable **xmlpath_processing**, all the tags within the path are searched. Tags that are not within the path cannot be searched. If **xmlpath_processing** is not enabled only individual tags can be searched.

Full paths and relative paths in path processing

The XML path can be either a full path or a relative path.

Full paths

Full paths begins with a slash (/). If you use the **all_xmltags** parameter with **xmlpath_processing**, all of the full paths are indexed. You can index specific full or relative paths when you use the **xmltags** parameter.

Given the XML fragment:

```
<text>
<book>
 <title>Graph Theory</title>
 <author>Stewart</author>
 <date>January 14, 2006</date>
</book>
<text>
```

The following full XML paths can be processed with the **xmlpath_processing** parameter:

```
/text/book/title
/text/book/author
/text/book/date
```

Tip: If you have indexed a full path, include the initial slash (/) in the search predicate. For example:

```
bts_contains("/text/book/author:stewart")
```

Relative paths

Relative paths begin with text. You can specify one or more relative or full paths with the **xmltags** parameter.

Based on the preceding XML fragment, each of the following relative XML paths can be processed with the **xmlpath_processing** parameter:

```
text/book/title
text/book/author
text/book/date
book/title
book/author
book/date
title
author
date
```

The field is created from the first matching path for the values specified with the **xmltags** parameter.

You can create an index for the book/title and the title fields:

```
CREATE INDEX books_bts ON books(xml_data bts_lvarchar_ops)
using bts(xmltags="(book/title,title)",xmlpath_processing="yes")
IN bts_sbspace;
```

In that case, the index will contain only the first matching field, book/title. It will not contain a title field:

```
book/title:Graph Theory
```

To view the fields that you have indexed, use the **bts_index_fields()** function. See "bts_index_fields() function" on page 20-1.

Example: Index XML paths

Use XML path processing to restrict searches by paths.

Given the XML fragment:

```
<boat>
 <skipper>Captain Black</skipper>
 <boatname>The Queen Anne's Revenge</boatname>
 <alternate>
  <skipper>Captain Blue Beard</skipper>
 </alternate>
</boat>
```

Following are the possible XML paths and text values:

```
/boat/skipper:Captain Black
/boat/boathame:The Queen Anne's Revenge
/boat/alterate/skipper:Captain Blue Beard
```

To create an index for boat/skipper and skipper, use the statement:

```
CREATE INDEX boats_bts ON boats(xml_data bts_lvarchar_ops)
using bts(xmltags="(boat/skipper,skipper)",xmlpath_processing="yes")
IN bts_sbspace;
```

Each path is compared to the values specified by the **xmltags** parameter. The index then creates fields for the entire first matching path found for each **xmltags** value. In this example, the first path matches `boat/skipper`. The third path matches `skipper`. The index will contain two fields that can be searched:

```
/boat/skipper:Captain Black
/boat/alterate/skipper:Captain Blue Beard
```

The include_contents index parameter

Use the **include_contents** parameter to add the `contents` field to the index.

The **include_contents** parameter must be used with either the **xmltags** parameter specified or with the **all_xmltags** or **all_xmlattrs** parameter enabled.

When you do not use XML index parameters, XML documents are indexed as unstructured text in the `contents` field. When you specify the **xmltags** parameter or you enable the **all_xmltags** parameter, you can add the `contents` field to the index by enabling the **include_contents** parameter. This allows you to search the unstructured text in the contents field in addition to fields containing the tag or attribute text.

To view the fields that you have indexed, use the **bts_index_fields()** function. See "bts_index_fields() function" on page 20-1.

Example: Index XML tag values and XML tag names

Use the **include_contents** parameter to search both XML tag values and XML tag names.

Given the XML fragment:

```
<book>
 <title>Graph Theory</title>
 <author>Stewart</author>
 <date>January 14, 2006</date>
</book>
```

To create a **bts** index for all the tags as well as the XML tags in their unstructured form, use the statement:

```
CREATE INDEX book_bts ON books(xml_data bts_lvarchar_ops)
USING bts(all_xmltags="yes",include_contents="yes")
IN bts_sbspace;
```

The index will have four fields; one for each of the XML tags and one for the `contents` field:

```
title:graph theory
author:stewart
date:january 14, 2006
contents:<book> <title>Graph Theory</title> <author>Stewart</author>
 <date>January 14, 2006</date> </book>
```

The strip_xmltags index parameter

Use the **strip_xmltags** parameter to add the untagged values to the `contents` field in the index. Attribute values are also removed.

Unlike other XML index parameters, you can use the **strip_xmltags** parameter in a CREATE INDEX statement without specifying the **xmltags** parameter or enabling the **all_xmltags** parameter. In this case, the contents field is created automatically.

However, if you specify the **xmltags** parameter or if you enable the **all_xmltags** parameter, you must also enable the **include_contents** parameter.

To view the fields that you have indexed, use the **bts_index_fields()** function. See "bts_index_fields() function" on page 20-1.

Example: Index XML tag values in a separate field

Given the XML fragment:

```
<book>
 <title>Graph Theory</title>
 <author>Stewart</author>
 <date>January 14, 2006</date>
</book>
```

To create an index with the untagged values only, use the statement:

```
CREATE INDEX books_bts ON books(xml_data bts_lvarchar_ops)
USING bts(strip_xmltags="yes") IN bts_sbspace;
```

The index will contain a single contents field:

```
contents:Graph Theory Stewart January 14, 2006
```

To create an index that has XML tag fields as well as a field for the untagged values, use the statement:

```
CREATE INDEX book_bts ON books(xml_data bts_lvarchar_ops)
USING bts(all_xmltags="yes",include_contents="yes",strip_xmltags="yes")
IN bts_sbspace;
```

The index will contain XML tag fields as well as the untagged values in the contents field:

```
title:graph theory
author:stewart
date:january 14, 2006
contents:Graph Theory Stewart January 14, 2006
```

The include_namespaces index parameter

Use the **include_namespaces** parameter to index XML tags that include namespaces in the qualified namespace format *prefix*:*localpart*. For example:

```
<book:title></book:title>
```

The **include_namespaces** parameter must be used with either the **xmltags** parameter specified or with the **all_xmltags** parameter enabled.

When you enable the **include_namespaces** parameter and the data includes the namespace in the indexed tags, you must use the namespace prefix in your queries and escape each colon (:) with a backslash (\).

For example, to search for the text Smith, in the field customer:name:, use the format:

```
bts_contains("/customer\:name:Smith")
```

To view the fields that you have indexed, use the **bts_index_fields()** function. See "bts_index_fields() function" on page 20-1.

Example: Index namespaces in XML data

The following XML fragment contains the namespace `book:title`:

```
<book>
<book:title>Graph Theory</book:title>
<author>Stewart</author>
<date>January 14, 2006</date>
</book>
```

You can create a **bts** index with the **`include_namespaces`** parameter disabled as in the statement:

```
CREATE INDEX books_bts ON books(xml_data bts_lvarchar_ops)
USING bts(all_xmltags="yes",include_namespaces="no",xmlpath_processing="yes")
IN bts_sbspace;
```

In that case, the namespace prefix `book:` is ignored. The index will have the following fields.

```
/book/title:graph theory
/book/author:stewart
/book/date:january 14, 2006
```

Also, you can create a **bts** index with the **`include_namespaces`** parameter enabled, as in the statement:

```
CREATE INDEX books_bts ON books(xml_data bts_lvarchar_ops)
USING bts(all_xmltags="yes",include_namespaces="yes",xmlpath_processing="yes")
IN bts_sbspace;
```

In that case, the tag with the namespace `book:title` is the first field. The index has the following fields:

```
/book/book:title:graph theory
/book/author:stewart
/book/date:january 14, 2006
```

To search the field `/book/book:title:` for the text `theory`, use the search predicate:

```
bts_contains("/book/book\:title:theory")
```

When you specify tags with the **`xmltags`** parameter, you can index the tags with and without namespaces in different combinations using the **`include_namespaces`** parameter. For example, given the XML fragments:

```
<bsns:bookstore>
  <title> Marine Buyers' Guide </title>
  <bns2:title> Boat Catalog </bns2:title>
 </bsns:bookstore>

<bsns:bookstore>
  <bns1:title> Toy Catalog </bns1:title>
  <bns2:title> Wish Book </bns2:title>
 </bsns:bookstore>
```

To index only the `title` tag, use the format:

```
CREATE INDEX bookstore_bts ON bookstores(xml_data bts_lvarchar_ops)
USING bts(xmltag="(title)",include_namespaces="yes")
IN bts_sbspace;
```

Even though the **include_namespaces** parameter is enabled, the index will contain only one field because the fields bns1:title and bns2:title do not match the specified tag title.

If you want to index a namespace, include the namespace prefix in the specified tags. For example if you use the format:

```
CREATE INDEX bookstore_bts ON bookstores(xml_data bts_lvarchar_ops)
USING bts(xmltag="(title,bns1:title)",include_namespaces="yes)
IN bts_sbspace;
```

The index will contain the fields:

```
title: Marine Buyers' Guide
bns1:title: Toy Catalog
```

The include_subtag_text index parameter

Use the **include_subtag_text** parameter to index XML tags and subtags as one string. The **include_subtag_text** parameter is useful when you want to index text that has been formatted with bold or italic <i></i> tags.

Use the **include_subtag_text** parameter with either the **xmltags** parameter specified or with the **all_xmltags** parameter enabled.

To view the fields that you have indexed, use the **bts_index_fields()** function. See "bts_index_fields() function" on page 20-1.

Example: Index subtags in XML data

You can use the **include_subtag_text** parameter to include the text within formatting tags in the indexed data.

Given the XML fragment:

```
<comment>
this
<bold> highlighted </bold>
text is very
<italic>
<bold>important</bold>
</italic>
to me
</comment>
```

If you create a **bts** index with the **include_subtag_text** parameter disabled:

```
CREATE INDEX comments_bts ON mylog(comment_data bts_lvarchar_ops)
USING bts(xmltags="(comment)",include_subtag_text="no") IN bts_sbspace;
```

The index will have three separate comment fields:

```
comment:this
comment:text is very
comment:to me
```

If you create a **bts** index with the **include_subtag_text** parameter enabled:

```
CREATE INDEX comments_bts ON mylog(comment_data bts_lvarchar_ops)
USING bts(xmltags="(comment)",include_subtag_text="yes") IN bts_sbspace;
```

All of the text is indexed in a single comment field:

```
comment:this highlighted text is very important to me
```

Chapter 19. Basic text search analyzers

A text analyzer prescribes how text is indexed.

A text analyzer converts input text into tokens that are indexed.

Analyzers differ in the ways that they process the following text attributes:
- Letter case
- Stopwords
- Chinese, Japanese, and Korean characters
- Numbers and non-alphabetic characters
- White spaces
- Word stems
- Word pronunciation

If your needs are different than any of the basic text search analyzers, you can create a user-defined analyzer.

analyzer index parameter

When you create a **bts** index, you can include the **analyzer** index parameter to set the default analyzer and any specific analyzers for specific fields.

The analyzer index parameter:

```
├──analyzer──=──"──┬─┬──────────┬─analyzer──────┬──"──────────────────────┤
                   │ └─field──:─┘               │
                   │  ┌─,──────────────────┐    │
                   │  ▼                    │    │
                   ├─(───┬──────────┬─analyzer─┴─)─┤
                   │     └─field──:─┘               │
                   ├─file──:──directory──/──filename──┤
                   └─table──:──table──.──column──────┘
```


Table 19-1. Options for the analyzer index parameter

Element	Description
analyzer	The name of the analyzer. Possible values: • **standard**: Default. Processes alphabetic characters, special characters, and numbers with stopwords. • **alnum**: Processes strings of numbers and characters into tokens. • **alnum+***characters*: Includes the specified characters in tokens. List characters without spaces. The maximum length of the character list is 128 bytes. • **cjk**: Processes Chinese, Japanese, and Korean text. Ignores surrogates. • **cjk.ws**: Processes Chinese, Japanese, and Korean text. Processes surrogates. • **esoundex**: Processes text into pronunciation codes. • **keyword**: Processes input text as a single token and adds trailing white spaces as necessary for fixed-length data type columns. • **keyword.rt**: Processes input text as a single token and removes trailing white spaces. • **simple**: Processes alphabetic characters only. Ignores stopword lists. • **snowball**: Processes text into stem words. • **snowball.***language*: Processes text into stem words in the specified language. • **soundex**: Processes text into pronunciation codes. • **stopword**: Processes alphabetic characters only, except stopwords. • **udr.***function_name*: Creates tokens according to the specified user-defined analyzer. • **whitespace**: Creates tokens that are based on white space only.
column	The name of the column that contains analyzer assignments.
directory	The path for the analyzer assignments file.
field	The XML tag, path, or the column name that is indexed.
filename	The name of the file that contains analyzer assignments.
table	The name of the table that contains analyzer assignments.

Usage

To use the same analyzer for all fields or columns that are indexed when you create the **bts** index, include the analyzer name without a field name. To use more than one analyzer, enclose multiple analyzer and field pairs in parentheses. To use one analyzer for most fields but other analyzers for specific fields, list the first analyzer without a field and the other analyzers with fields. The first analyzer is used for all fields except the ones that are explicitly listed with analyzer assignments.

You can specify the list of analyzers by field in a table column or in a file. The file or table must be readable by the user who creates the index. Separate the field name and analyzer pairs in the file or table by commas, white spaces, new lines, or a combination of those separators. The file or table becomes read-only when the index is created. If you want to add or change analyzer assignments, you must drop and re-create the index.

Examples

The following example creates a **bts** index on one column and uses the CJK analyzer:

```
CREATE INDEX desc_idx ON products (brands  bts_char_ops)
 USING bts (analyzer="cjk") IN sbsp1;
```

The following example creates a **bts** index on two XML fields and uses a different analyzer for each field:

```
CREATE INDEX boats_bts
ON boats(xml_data bts_lvarchar_ops)
USING bts
(
xmltags="(skipper,boatname)" ,
analyzer="(skipper:soundex,boatname:snowball)"
)
IN bts_sbspace;
```

Analyzer support for query and index options

The basic text search analyzer that you specify affects whether you can use stopwords or a thesaurus when you create an index and which query term modifiers you can use when you query text.

The following table shows which analyzers support query term modifiers, lowercase processing, stopwords, and a thesaurus.

Table 19-2. Analyzers and query term modifiers and index parameters

Analyzer	Word	Phrase	Wildcard	Fuzzy	Proximity	Range	Boolean	Lowercase	Stopwords	Thesaurus
Alnum	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
CJK	yes	yes	1	1	1	1	yes	yes	yes	yes
eSoundex	yes	yes	2	2	no	no	yes	yes	yes	no
Keyword	yes	yes	yes	yes	yes	no	yes	no	no	no
Simple	yes	yes	yes	yes	yes	yes	yes	yes	no	yes
Soundex	yes	yes	2	2	no	no	yes	yes	yes	no
Snowball	3	3	4	4	4	3	yes	yes	yes	yes
Standard	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
Stopword	yes	yes	yes	yes	yes	yes	yes	yes	yes	yes
User-defined analyzer	yes	5	5	5	5	5	yes	yes	yes	yes
Whitespace	yes	yes	yes	yes	yes	yes	yes	no	no	yes

1 = ISO Latin characters are supported.

2 = Must use the Soundex or eSoundex codes in the search terms.

3 = Depends on the stem word.

4 = The patterns must be on the stem word. The operation works on the stem word.

5 = Depends on the user-defined analyzer code.

Alnum analyzer

The Alnum analyzer is useful if you want to index words that contain numbers and other characters.

The Alnum analyzer processes text in the following ways:
- Indexes numbers as part of the word.
- Does not index stopwords.
- Converts alphabetic characters to lowercase.
- Treats as white space all non-alphanumeric characters unless the characters are included in the *characters* list. Non-alphanumeric characters include: #, %, $, @, &, :, ', (,), -, _, \, and /.

Include a list of characters to index as part of words by using the **alnum+***characters* syntax. List characters without spaces. The maximum length of the character list is 128 bytes.

Examples

In these examples, the input string is shown on the first line and the resulting tokens are shown on the second line, each surrounded by square brackets.

In the following example, words that contain both numbers and letters are indexed together and special characters are treated as white spaces:

```
1002A 3234 abc123 xyz-abc lmn_opq
[1002a] [3234] [abc123] [xyz] [abc] [lmn] [opq]
```

In the following example, the **analyzer** index parameter is set to **alnum+_-**. The hyphen and underscore characters are indexed as part of words:

```
1002A 3234 abc123 xyz-abc lmn_opq
[1002a] [3234] [abc123] [xyz-abc] [lmn_opq]
```

CJK analyzer

The CJK analyzer processes Chinese, Japanese, and Korean characters into tokens that are indexed.

The CJK analyzer processes text characters in the following ways:
- Transforms the character sets to UTC-4. Half-width and full-width forms are converted so that they have equivalent characters. For example, fullwidth_digit_zero and digit_zero are treated as the same character.
- Indexes Chinese, Japanese, and Korean characters in overlapping pairs.
- Indexes Latin alphabetic, numeric, and the special characters _, +, and #.
- Stopwords are not indexed.
- Does not process supplementary code points if the analyzer name is **cjk**,
- Processes supplementary code points as surrogate pairs if the analyzer name is **cjk.ws**,

Examples

In the following example, the first line shows the input string, in which C1, C2, C3 and C4 represent Chinese, Japanese, or Korean characters. The second line shows the resulting tokens, each surrounded by square brackets:

```
sailC1C2C3C4boat
[sail] [C1C2] [C2C3] [C3C4] [boat]
```

eSoundex analyzer

The eSoundex, or Extended Soundex, analyzer uses the Soundex algorithm to convert words into codes based on the English pronunciation of their consonants.

Vowel sounds are not included unless the vowel is the first letter of the word. The eSoundex analyzer is the similar to the Soundex analyzer except that it allows fewer or greater than four characters in its codes, depending on the length of the word. The eSoundex analyzer is useful if you want to search text based on how words sound. Because the text is converted to codes, you cannot perform proximity and range searches or specify a thesaurus.

The eSoundex analyzer processes text characters in the following ways:

- Stopwords are not indexed.
- Numbers and special characters are ignored.
- The colon (:) character is treated as a whitespace, so that characters on either side of it are considered separate words.

Examples

In these examples, the input string is shown on the first line and the resulting tokens are shown on the second line, each surrounded by square brackets.

In the following example, the words "the" are not converted to tokens because they are stopwords and the rest of the words are converted to eSoundex codes that begin with the first letter of the word:

```
The Quick Brown Fox Jumped Over The Lazy Dog
[q2] [b65] [f2] [j513] [o16] [l2] [d2]
```

In the following example, the colon is treated as a whitespace and the backslash is ignored:

```
c:/informix
[c] [i51652]
```

In the following example, the ampersand is ignored:

```
XY&Z Corporation
[x2] [c61635]
```

In the following example, the e-mail address is considered one word:

```
xyz@example.com
[x2251425]
```

In the following example, numbers are ignored:

```
1abc 12abc abc1 abc12
[a12] [a12] [a12] [a12]
```

In the following examples, three words with the same stem word have different codes:

```
accept
[a213]
acceptable
[a21314]
acceptance
[a21352]
```

Keyword analyzer

The Keyword analyzer converts input text into a single token without alteration.

The Keyword analyzer is useful if you want to index single words exactly as they are, however, any type of input text is indexed. You cannot search a range or specify a thesaurus on text indexed by the Keyword analyzer.

The Keyword analyzer processes text characters in the following ways:

- Stopword lists are ignored. All words are indexed.
- Alphabetic characters are not converted to lowercase.
- Numeric and special characters are indexed.
- White spaces are indexed. Queries for text that includes white spaces must escape each white space by a backslash (\) character.
- If the analyzer name is **keyword.rt**, removes trailing white spaces during indexing and querying.
- If the analyzer name is **keyword**, indexes trailing white spaces.
  - For indexed columns that have fixed-length data types, the **keyword** analyzer adds white spaces as necessary to reach the column length. For example, if the text column is of type CHAR(6) and a string has three characters, `abc`, the string is indexed with three trailing white spaces, regardless of whether the string included one or more trailing white spaces: `abc   `. Queries require the correct number of escaped trailing white spaces: for example, `abc\ \ \ `.
  - For indexed columns that have variable-length data types, any trailing white spaces that are included in the string are indexed. For example, if the text column is of type LVARCHAR, the string abc with one trailing white space is indexed as a different token from the string abc with two trailing white spaces. Queries require the correct number of escaped trailing white spaces: for example, `abc\ ` or `abc\ \ `.

Examples

In these examples, the input string is shown on the first line and the resulting tokens are shown on the second line, each surrounded by square brackets. The following examples show that the entire input string is preserved exactly as is:

```
The Quick Brown Fox Jumped Over The Lazy Dog
[The Quick Brown Fox Jumped Over The Lazy Dog]

-12 -.345 -898.2 -56. -
[-12 -.345 -898.2 -56. -]

XY&Z Corporation
[XY&Z Corporation]

xyz@example.com
[xyz@example.com]
```

The following query string searches for the string `The Quick Brown Fox Jumped Over The Lazy Dog`:

```
'The\ Quick\ Brown\ Fox\ Jumped\ Over\ The\ Lazy\ Dog'
```

Simple analyzer

The Simple analyzer converts text to tokens that contain only alphabetic characters.

The Simple analyzer is useful if you want to index every word and ignore non-alphabetical characters.

The Simple analyzer processes text characters in the following ways:
- Each word is processed into a separate token.
- Alphabetic characters are converted to lowercase.
- Numeric and special characters are treated as white spaces.
- Stopword lists are ignored. All words are indexed.

Because the Simple analyzer does not support stopwords, omit the word TO from range queries.

Examples

In these examples, the input string is shown on the first line and the resulting tokens are shown on the second line, each surrounded by square brackets.

In the following example, every word is converted to a lowercase token:

```
The Quick Brown Fox Jumped Over The Lazy Dog
[the][quick][brown][fox] [jumped] [over] [the] [lazy] [dog]
```

In the following example, the @ symbol and period are treated as white spaces:

```
xyz@example.com
[xyz] [example] [com]
```

In the following example, numbers are not included in the tokens:

```
1abc 12abc abc1 abc12
[abc] [abc] [abc] [abc]
```

Soundex analyzer

The Soundex analyzer uses the Soundex algorithm to convert words into four-character codes based on the English pronunciation of their consonants.

Vowel sounds are not included unless the vowel is the first letter of the word. Additional sounds beyond the first four phonetic sounds are ignored. If a word has fewer than four phonetic sounds, zeros are used to complete the four-character codes. The Soundex analyzer is the similar to the eSoundex analyzer except that it uses four characters in its codes, regardless of the length of the word. The Soundex analyzer is useful if you want to search text based on how the beginnings of words sound. Because the text is converted to codes, you cannot perform proximity and range searches or specify a thesaurus.

The Soundex analyzer processes text characters in the following ways:
- Stopwords are not indexed.
- Numbers and special characters are ignored.

- The colon (:) character is treated as a whitespace, so that characters on either side of it are considered separate words.

Examples

In these examples, the input string is shown on the first line and the resulting tokens are shown on the second line, each surrounded by square brackets. All codes consist of four characters.

In the following example, the words "the" are not converted to tokens because they are stopwords and the rest of the words are converted to Soundex codes that begin with the first letter of the word:

```
The Quick Brown Fox Jumped Over The Lazy Dog
[q200] [b650] [f200] [j513] [o160] [l200] [d200]
```

In the following example, the colon is treated as a whitespace and the backslash is ignored:

```
c:/informix
[c000] [i516]
```

In the following example, the ampersand is ignored:

```
XY&Z Corporation
[x200] [c616]
```

In the following example, the e-mail address is considered one word:

```
xyz@example.com
[x225]
```

In the following example, numbers are ignored:

```
1abc 12abc abc1 abc12
[a120] [a120] [a120] [a120]
```

In the following examples, three words with the same stem word have the same code:

```
accept
[a213]
acceptable
[a213]
acceptance
[a213]
```

Snowball analyzer

The Snowball analyzer converts words into language and code set specific stem words.

The Snowball analyzer is similar to the Standard analyzer except that is converts words to stem words.

The Snowball analyzer processes text characters in the following ways:

- Converts words to stem word tokens.
- Stopwords are not indexed.
- Converts alphabetical characters to lower case.
- Ignores colons, #, %, $, parentheses, and slashes.

- Indexes underscores, hyphens, @, and & symbols when they are part of words or numbers.
- Separately indexes numbers and words if numbers appear at the beginning of a word.
- Indexes numbers as part of the word if they are within or at the end of the word.
- Indexes apostrophes if they are in the middle of a word, but removes them if they are at the beginning or end of a word.
- Ignores an apostrophe followed by the letter s at the end of a word.

By default, the Snowball analyzer uses the language and code set that is specified by the **DB_LOCALE** environment variable. You can specify a different language for the Snowball analyzer by appending the language name or synonym to the Snowball analyzer name in the CREATE INDEX statement: **snowball.***language*.

The Snowball analyzer supports the following language names and synonyms that belong to the 8859-1 or UTF-8 code sets:
- Danish, da, dan
- Dutch, nl nld, dut
- English, en, eng
- Porter, por (the original English stemmer)
- Finnish, fi, fin
- French, fr, fra, fre
- German, de, deu, ger
- Italian, it, ita
- Norwegian, no, nor
- Portuguese, pt
- Spanish, es, esl, spa
- Swedish, sv, swe

The Snowball analyzer supports the following language name and synonyms that belong to the KOI-8 or UTF-8 code sets: Russian, ru, rus.

Examples

In these examples, the input string is shown on the first line and the resulting tokens are shown on the second line, each surrounded by square brackets. These examples use the English language, specified by the **analyzer="snowball.en"** index parameter. For examples of how the Snowball analyzer uses word stemming in languages other than English, see the Snowball web site at http://snowball.tartarus.org.

In the following example, stopwords are removed, the words are converted to lower case, and the word "lazy" is converted to its stem word:

```
The Quick Brown Fox Jumped Over The Lazy Dog
[quick] [brown] [fox] [jump] [over] [lazi] [dog]
```

In the following example, the apostrophe at the beginning of a word and the apostrophe followed by an s are ignored, but the apostrophe in the middle of a word is indexed:

```
Prequ'ile Mark's 'cause
[prequ'ile] [mark] [cause]
```

In the following example, the colon and backslash are ignored:

```
c:/informix
[c] [informix]
```

In the following example, the ampersand is indexed as part of the company name:

```
XY&Z Corporation
[xy&z] [corpor]
```

In the following example, the e-mail address is indexed as is:

```
xyz@example.com
[xyz@example.com]
```

In the following example, the three different words are indexed with the same stem word:

```
accept
[accept]

acceptable
[accept]

acceptance
[accept]
```

Standard analyzer

The Standard analyzer removes stopwords and indexes words, numbers, and some special characters. The Standard analyzer is the default analyzer.

The Standard analyzer processes text characters in the following ways:

- Stopwords are not indexed.
- Converts alphabetical characters to lower case.
- Ignores colons, #, %, $, parentheses, hyphens, and slashes.
- Indexes underscores, @, and & symbols when they are part of words or numbers.
- Separately indexes number and words if numbers appear at the beginning of a word.
- Indexes numbers as part of the word if they are within or at the end of the word.
- Indexes apostrophes if they are in the middle of a word, but removes them if they are at the beginning or end of a word.
- Ignores an apostrophe followed by the letter s at the end of a word.

Examples

In these examples, the input string is shown on the first line and the resulting tokens are shown on the second line, each surrounded by square brackets.

In the following example, stopwords are removed and the words are converted to lower case:

```
The Quick Brown Fox Jumped Over The Lazy Dog
[quick] [brown] [fox] [jumped] [over] [lazy] [dog]
```

In the following example, the apostrophe at the beginning of a word and the apostrophe followed by an s are ignored, but the apostrophe in the middle of a word is indexed:

```
Prequ'ile Mark's 'cause
[prequ'ile] [mark] [cause]
```

In the following example, the colon and backslash are ignored:

```
c:/informix
[c] [informix]
```

In the following example, the ampersand is indexed as part of the company name:

```
XY&Z Corporation
[xy&z] [corporation]
```

In the following example, the e-mail address is indexed as is:

```
xyz@example.com
[xyz@example.com]
```

In the following example, numbers at the beginning of the words are separated into different tokens, while numbers at the end of words are included in a single token:

```
1abc 12abc abc1 abc12
[1] [abc] [12] [abc] [abc1] [abc12]
```

Stopword analyzer

The Stopword analyzer removes stopwords and converts text to tokens that contain only alphabetic characters.

The Stopword analyzer is useful if you want to remove stopwords and ignore non-alphabetical characters.

The Stopword analyzer processes text characters in the following ways:

- Each word is processed into a separate token.
- Alphabetic characters are converted to lowercase.
- Numeric and special characters are treated as white spaces.
- Stopwords are not indexed.

Examples

In these examples, the input string is shown on the first line and the resulting tokens are shown on the second line, each surrounded by square brackets.

In the following example, stopwords are removed and the letters are converted to lowercase:

```
The Quick Brown Fox Jumped Over The Lazy Dog
[quick] [brown] [fox] [jumped] [over] [lazy] [dog]
```

In the following example, the @ symbol and period are treated as white spaces:

```
xyz@example.com
[xyz] [example] [com]
```

In the following example, numbers are not included in the tokens:

```
1abc 12abc abc1 abc12
[abc] [abc] [abc] [abc]
```

User-defined analyzer

A user-defined analyzer processes text into tokens according to a user-defined function.

You can write a user-defined function to process text into tokens according to your needs. Use **udr.***function_name* as the analyzer name with the **analyzer** option when you create a basic text search index.

Examples

The following function, which is written in C, processes alphabetical and numeric characters into tokens and ignores all special characters except underscore (_):

```
/*ARGSUSED*/
UDREXPORT
mi_lvarchar* tokenize_alnum(
    mi_lvarchar*    string,
    MI_FPARAM*      fparam)
{
    mi_integer      status = MI_OK;
    mi_lvarchar*    rtn = NULL;
    gl_mchar_t*     src = NULL;
    gl_mchar_t*     tgt = NULL;
    mi_integer      token = 0;
    gl_mchar_t*     s;
    gl_mchar_t*     r;

    ifx_gl_init();
    if (((src = (gl_mchar_t*)mi_lvarchar_to_string(string)) == NULL) ||
        ((tgt = (gl_mchar_t*)mi_alloc((strlen(src)*4)+1)) == NULL)) {
        status = MI_ERROR;
        goto cleanup;
    }
    s = src;
    r = tgt;
    while ((s != NULL) && (*s != '\0')) {
        if ((ifx_gl_ismalnum(s, IFX_GL_NO_LIMIT)) || (*s == '_')) {
            if (!token) {
                if (r != tgt) *r++ = ' ';
                *r++ = '[';
                token = 1;
            }
            ifx_gl_mbsncpy(r, s, IFX_GL_NULL, 1);
            r = ifx_gl_mbsnext(r, IFX_GL_NO_LIMIT);
        }
        else {
            if (token) {
                *r++ = ']';
                token = 0;
            }
        }
        s = ifx_gl_mbsnext(s, IFX_GL_NO_LIMIT);
    }
    if (token) *r++ = ']';
    *r = '\0';
    if ((rtn = mi_string_to_lvarchar((char*)tgt)) == NULL) {
        status = MI_ERROR;
        goto cleanup;
    }
cleanup:
    if ((status != MI_OK) &&
        (rtn != NULL)) {
        mi_var_free(rtn);
        rtn = NULL;
```

```
    }
    if (tgt != NULL) mi_free(tgt);
    if (src != NULL) mi_free(src);
    if (rtn == NULL) mi_fp_setreturnisnull(fparam, 0, MI_TRUE);
    return rtn;
}
```

The following statement registers the function so that the database server can use it:

```
CREATE FUNCTION tokenize_alnum (lvarchar)
    RETURNS lvarchar
    WITH (NOT VARIANT)
    EXTERNAL NAME "$INFORMIXDIR/extend/myblade/myblade.bld(tokenize_alnum)"
        LANGUAGE C;
```

When an index is created with the **analyzer="udr.tokenize_alnum"** option, the following example shows that no special characters except the underscore are indexed:

```
quick! #$%&^^$## Brown fox under_score
[quick] [Brown] [fox] [under_score]
```

Whitespace analyzer

The Whitespace analyzer processes characters into tokens based on whitespaces. All characters between whitespaces are indexed without alteration.

The Whitespace analyzer processes text characters in the following ways:
- Stopword lists are ignored. All words are indexed.
- Does not change letter case.
- Indexes numbers and special characters.

Because the Whitespace analyzer does not support stopwords, omit the word TO from range queries.

Examples

In the following examples, the input text is shown on the first line and the resulting indexed tokens, which are surrounded by square brackets, are shown on the second line.

In the following example, all words are indexed exactly as they are:

```
The Quick Brown Fox Jumped Over The Lazy Dog
[The] [Quick] [Brown][Fox] [Jumped][Over] [The] [Lazy] [Dog]
```

In the following example, all numbers and special characters are indexed:

```
-12 -.345 -898.2 -56. –
[-12] [-.345] [-898.2] [-56.] [-]
```

In the following example, the e-mail address is indexed as one token:

```
xyz@example.com
[xyz@example.com]
```

Chapter 20. Basic text search functions

You can use basic text search functions to provide information about **bts** indexes, compact bts indexes, and configure tracing.

bts_index_compact() function

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

Syntax

```
►►—bts_index_compact—(—'—index_name—'—)————————————————————►◄
```

index_name
The name of the **bts** index for which you want to delete rows.

Usage

Use the **bts_index_compact()** function to delete documents from a **bts** index that was created with the default deletion mode parameter of **`delete="deferred"`**. The **bts_index_compact()** function releases space in the index by immediately deleting the rows marked as deleted. The index is unavailable while it is rewritten. Optionally, you can include the index storage space path and file name, the database name, and the owner name in addition to the index name, separated by forward slash (/) characters.

Documents marked as deleted can also be deleted with the **oncheck** utility. For **oncheck** syntax and information about optimizing the **bts** index, see "delete index parameter" on page 15-11.

Return codes

t The operation was successful.

f The operation was unsuccessful.

Example

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

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

bts_index_fields() function

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

Syntax

```
►►—bts_index_fields—(—'—index_name—'—)————————————————————►◄
```

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

Usage

Use the **bts_index_fields()** function to identify searchable fields in the **bts** index. Optionally, you can include the index storage space path and file name, the database name, and the owner name in addition to the index name, which is separated by forward slash (/) characters.

The **bts_index_fields()** function returns one default field that is called `contents` unless any of the following conditions are true:

- The index is a composite index that has each column that is indexed separately because the index definition includes the **query_default_field="*"** index parameter. The **bts_index_fields()** function returns the names of the indexed columns.
- The index contains XML tags because the index definition includes the **`all_xmltags`** or **`xmltags`** index parameter. The **bts_index_fields()** function returns the indexed tags. If the **include_contents** index parameter is included in the index definition, the **bts_index_fields()** function also returns the `contents` field.
- The index contains JSON field name-value pairs because the index definition includes the **all_json_names** or **json_names** index parameter. The **bts_index_fields()** function returns the indexed field names. If the **include_contents** index parameter is included in the index definition, the **bts_index_fields()** function also returns the `contents` field.

When you specify tags with the **`xmltags`** parameter, the **bts_index_fields()** function returns only field names for tags that exist in the indexed column. However, if later you add a row that contains the specified tag name, the field name for that tag appears in the output.

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

Example: Unstructured index

The following statement creates an unstructured index:

```
CREATE INDEX desc_idx ON products (brands  bts_char_ops)
 USING bts IN sbsp1;
```

The **bts_index_fields()** function returns the default field: `contents`.

Examples: Structured indexes on an XML document

These examples are based on the following XML fragment:

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

The following statement indexes the specified XML tags:

```
CREATE INDEX boats_bts ON boats(xml_data bts_lvarchar_ops)
USING bts(xmltags="(skipper,boatname,crew)") IN bts_sbspace;
```

The **bts_index_fields()** function returns the following field names:

```
boatname
skipper
```

The field name for the tag `crew` is not returned because it does not exist in the XML fragment example.

The following statement indexes all tags and paths:

```
CREATE INDEX boats_bts ON boats(xml_data bts_lvarchar_ops)
USING bts(all_xmltags="yes",xmlpath_processing="yes")
IN bts_sbspace;
```

The **bts_index_fields()** function returns field names that include full paths:

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

The following statement indexes all tags and includes the `contents` field:

```
CREATE INDEX boats_bts ON boats(xml_data bts_lvarchar_ops)
USING bts(all_xmltags="yes",include_contents="yes")
IN bts_sbspace;
```

The **bts_index_fields()** function returns the following fields:

```
boatname
contents
skipper
```

Examples: Structured indexes on a JSON document

These examples are based on the following JSON document:

```
{ "person" : {
     "givenname" : "Jim",
     "surname" : "Flynn",
     "age" : 29,
     "cars" : [ "dodge", "olds" ],
     "parents":[
        { "givenname" : "Slim",
          "surname" : "Flynn",
          "surname" : "Kim" }
     ]
}
```

The following statement indexes all field name-value pairs:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(all_json_names="yes");
```

The **bts_index_fields()** function returns the following fields:

```
age
cars
givenname
surname
```

The following statement indexes all field name-value pairs and includes the `contents` field:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(all_json_names="yes",
                 include_contents="yes");
```

The **bts_index_fields()** function returns the following fields:

```
age
cars
contents
givenname
surname
```

The following statement indexes a single field:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(json_names="surname");
```

The **bts_index_fields()** function returns the following field:

```
surname
```

The following statement indexes the specified field names and paths:

```
CREATE INDEX bts_idx
          ON json_tab (docs bts_json_ops)
       USING bts(
                 json_names="(parents.surname,
                              person.givenname)",
                 json_path_processing="yes"
        );
```

The **bts_index_fields()** function returns the following fields:

```
person.given name
parents.surname
```

Related concepts:

"The all_xmltags index parameter" on page 18-4

"The xmltags index parameter" on page 18-2

Related reference:

"Basic text search index fields" on page 16-3

"all_json_names index parameter" on page 17-4

"json_names index parameter" on page 17-8

bts_release() function

The **bts_release()** function provides the internal release version number of the basic text search engine.

Syntax

```
►►—bts_release—()——————————————————————————————————————————————►◄
```

Usage

Use the **bts_release()** function if IBM Software Support asks you for the basic text search version number.

Return codes

This function returns the name and release version number of the basic text search engine.

Example

Example output:

```
BTS 3.00 Compiled on Wed Jan 19 11:25:52 CDT 2011
```

bts_tracefile() function

The **bts_tracefile()** function specifies the location where the trace file is written. Use this function together with the **bts_tracelevel()** function to trace basic text search-related events.

Syntax

```
►►──bts_tracefile──(──filename──)──────────────────────────────────►◄
```

filename
: The full path and name of the file to which trace information is appended. The file must be writable by user **informix**. If no file name is provided, a standard `session_id.trc` file is placed in the `$INFORMIXDIR/tmp` directory.

Usage

Use the **bts_tracefile()** function to troubleshoot events related to the basic text searches.

For the syntax for **bts_tracelevel()**, see "bts_tracelevel() function."

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');
```

bts_tracelevel() function

The **bts_tracelevel()** function sets the level of tracing. Use this function together with the **bts_tracefile()** function to trace basic text search-related events.

Syntax

```
►►──bts_tracelevel──(──level──)────────────────────────────────────►◄
```

level The level of tracing output:

1 UDR entry points.

10 UDR entry points and lower-level calls.

20 Trace information and small events.

100 Memory resource tracing (very verbose).

If you enter a value from 1-9, it is treated as level 1, a value between 10 and 19 is treated as level 10, a value between 20 and 99 is treated as level 20. A value greater than or equal to 100 is treated as level 100.

Usage

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

For the syntax for **bts_tracefile()**, see “bts_tracefile() function” on page 20-5.

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 21. Basic text search performance

The performance of basic text search queries depends on the **bts** index, disk space, configuration parameters, and BTS virtual processors.

When you create a **bts** index, include the following index parameters to improve the performance of basic test search queries:

- Set the **delete** index parameter to remove index information for deleted documents and release disk space. You can optimize the index manually or automatically after every delete operation.
- Set the **query_default_field=*** index parameter to create a composite index. When you run a basic text search query, the text from all the columns is searched in the `contents` field as if it was one string. This method can result in better query performance than using a UNION clause to combine the results of multiple queries on multiple **bts** indexes.

Include the following index parameters to improve the performance of building the **bts** index:

- Set the **tempspace** index parameter to the name of a temporary sbspace. Building the index in a temporary sbspace is faster than building the index in an sbspace that logs transactions.
- Set the **xact_ramdirectory=yes** index parameter to build the **bts** index in memory. Building the index in memory can be faster than building the index in a temporary sbspace.

The **bts** index is in an sbspace. If you defined multiple BTS virtual processors, each one can simultaneously process a different transaction. However, when a transaction that contains INSERT, DELETE, or UPDATE statements that affect the **bts** index is being committed, the transaction acquires an exclusive lock on the **bts** index. Any other concurrent transaction waits for up to 15 minutes for the lock to be released.

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

Related concepts:

"query_default_field index parameter" on page 15-13

Related tasks:

"Creating a space for temporary data" on page 15-4

Related reference:

"delete index parameter" on page 15-11

"xact_ramdirectory index parameter" on page 15-19

Disk space for the bts index

The size of the external **bts** index depends on the number of documents being indexed as well as the number of words and the number of unique words in those documents.

If you receive an I/O error such as `(BTSA1) - bts clucene error: IO error: File IO Write error`, check the online log. The probable cause is insufficient disk space. If this happens, drop the **bts** index with a DROP INDEX statement and recreate it on a disk with enough disk space.

To prevent running out of space for the **bts** index, create a dedicated sbspace for the **bts** index and a separate sbspace for temporary data. A separate sbspace for temporary data might also improve the speed of creating and updating the **bts** index.

See Chapter 15, "Preparing for basic text searching," on page 15-1 for the procedure to create a **bts** index. See the *IBM Informix Guide to SQL: Syntax* for instructions for the DROP INDEX statement.

Adding BTS virtual processors to run multiple queries simultaneously

You can increase the number of basic text search queries or other index operations that can run at the same time by adding additional BTS virtual processors.

Each Basic Text Search function, including **bts_contains()**, runs in a BTS virtual processor without yielding. If basic text search queries are slow because multiple users are running queries at the same time, you can add more BTS virtual processors so that queries run simultaneously, each in their own virtual processor.

To dynamically add BTS virtual processors for the current database server session:

Run the **onmode -p** command, specifying the number of virtual processors to add and the BTS virtual processor class. For example, the following command adds three BTS virtual processors: `onmode -p 3 bts`

Alternatively, you can use the SQL administration API **task()** or **admin()** function with the **`onmode`** and **`p`** arguments to add BTS virtual processors.

To permanently increase the number of BTS virtual processors, set the value of the **VPCLASS bts** configuration parameter in the `onconfig` file and then restart the database server. If the `onconfig` file contains an existing entry for the **VPCLASS bts** configuration parameter, update that entry; otherwise, add a new entry for the **VPCLASS bts** configuration parameter.

For more information about the **onmode** utility or the SQL administration API, see the *IBM Informix Administrator's Reference*,

Tune configuration parameters for basic text searching

You can optimize the performance of basic text searches by tuning certain configuration parameters.

AUTO_READAHEAD
The AUTO_READAHEAD configuration parameter enables automatic read-ahead. Pages that are brought into the bufferpool cache during sequential scans of data records can improve the performance of a query when the server detects that the query is encountering I/O.

BUFFERPOOL
The BUFFERPOOL configuration parameter defines a buffer pool for pages that correspond to each unique page size that is used by your dbspaces. You can specify information about the buffer pool, including its size, the number of

LRU queues in the buffer pool, the number of buffers in the buffer pool, and minimum and maximum percentages of modified pages in the LRU queues.

RESIDENT
The RESIDENT configuration parameter specifies whether the resident portion of shared memory remains resident in operating system physical memory. If your operating system supports forced residency, you can improve the performance of searches by specifying that the resident portion of shared memory is not swapped to disk. Set the RESIDENT configuration parameter to 1 (on).

VPCLASS
You can add the **noage** option to the VPCLASS configuration parameter setting for the BTS virtual processor. The **noage** option improves performance by disabling priority aging by the operating system. The BTS virtual processor that is created automatically does not include the **noage** option.

Related concepts:

The RESIDENT configuration parameter and memory utilization (Performance Guide)

The BUFFERPOOL configuration parameter and memory utilization (Performance Guide)

Related reference:

AUTO_READAHEAD configuration parameter (Administrator's Reference)

VPCLASS configuration parameter (Administrator's Reference)

Chapter 22. Basic text search error codes

Basic text searching has specific error messages.

The following table lists error codes for basic text searching.

SQLstate	Description
BTS01	bts error, assertion failed. File %FILE%, line %LINE%
BTS02	bts internal error. File %FILE%, line %LINE%
BTS03	bts error - could not set trace level to %PARAM1% for trace class %PARAM2%
BTS04	bts error - could not set trace output file to %PARAM1%
BTS05	bts error - unique index not supported
BTS06	bts error - cluster index not supported
BTS08	bts error - cannot query the table %TABLENAME%
BTS09	bts error - BTS index only supports extspaces and sbspaces
BTS10	bts error - cannot get connection descriptor
BTS11	bts error - extspace not specified
BTS12	bts error - cannot determine index owner
BTS13	bts error - cannot determine index name
BTS14	bts error - cannot create directory %PARAM1%
BTS15	bts error - current vpclass (%VPCLASS%) is not specified as noyield
BTS16	bts error - too many virtual processors running (%NUMVPS%) for the current vpclass (%VPCLASS%), 1 is the maximum
BTS17	bts error - out of memory
BTS18	bts error - SQL Boolean expression are not supported with bts_contains
BTS19	bts error - cannot query with a null value
BTS20	bts error - invalid value for index delete parameter: %PARAM1% should be either immediate or deferred
BTS21	bts error - unsupported type: %PARAM1%
BTS22	bts error - bts_contains requires an index on the search column
BTS23	bts error - cannot register end-of-transaction-callback
BTS24	bts error - invalid value for %PARAM1% parameter: %PARAM2% should be an integer value greater than 0
BTS25	bts error - CLOB or BLOB is too large, must be less than or equal to 2,147,483,647 bytes
BTS26	bts error - clob or blob is too large, must be less than or equal to 2,147,483,647
BTS27	bts error - BTS indexes in external spaces only permitted on primary or standard servers
BTS28	bts error - invalid value for the %PARAM1% parameter: %PARAM2% should be "unlimited" or an integer value greater than 0
BTS29	bts error - invalid value for the %PARAM1% parameter: %PARAM2% should be either "and" or "or"
BTS30	bts error - invalid value for the PARAM1% parameter: %PARAM2% should be either yes or no
BTS31	bts error - invalid value for the %PARAM1% parameter: %PARAM2% should be either yes, yes_with_tag or no

SQLstate	Description
BTS32	bts error - invalid value for the %PARAM1% parameter: %PARAM2% should be either yes, yes_with_database_name or no
BTS33	bts error - incorrect value for the %PARAM1% parameter: %PARAM2% should be either yes, yes_with_positions, yes_with_offsets or no
BTS34	bts error - uppercase characters are not allowed in stopwords
BTS35	bts internal error - mi_open() failed. File %FILE%, line %LINE%
BTS36	bts internal error - mi_lo_open() failed. File %FILE%, line %LINE%
BTS37	bts internal error - mi_lo_seek() failed. File %FILE%, line %LINE%
BTS38	bts internal error - mi_lo_read() failed. File %FILE%, line %LINE%
BTS39	bts internal error - ifx_int8toasc() failed. File %FILE%, line %LINE%
BTS40	bts internal error - mi_lo_spec_init() failed. File %FILE%, line %LINE%
BTS41	bts internal error - mi_lo_create() failed. File %FILE%, line %LINE%
BTS42	bts internal error - mi_lo_increfcount() failed. File %FILE%, line %LINE%
BTS43	bts internal error - ifx_int8cvlong() failed. File %FILE%, line %LINE%
BTS44	bts internal error - mi_lo_write() failed. File %FILE%, line %LINE%
BTS45	bts error - cannot open file %FILENAME%
BTS46	bts error - cannot create file %FILENAME%
BTS47	bts error - xml syntax error
BTS48	bts error - invalid hex specification: \x%PARAM1%%PARAM2%
BTS49	bts error - the GLS character name '%PARAM1%' is not found
BTS50	bts error - if either xmltags is specified or all_xmltags is enabled, then include_contents must be enabled if strip_xmltags is enabled
BTS51	bts error - xmlpath_processing cannot be enabled unless either xmltags is specified or all_xmltags is enabled.
BTS52	bts error – parameter %PARAM1% and %PARAM2% parameters are mutually exclusive
BTS53	bts error - invalid value for the %PARAM1% parameter: %PARAM2% should be a lower value
BTS54	bts error - cannot write to file %FILENAME%
BTS55	bts error - cannot read from file %FILENAME%
BTS56	bts error - bad magic number on file %FILENAME%
BTS57	bts error - the specified table (%TABLENAME%) is not in the database
BTS58	bts error - column (%COLUMNNAME%) not found in specified table (%TABLENAME%)
BTS59	bts error - column (%COLUMNNAME%) in specified table (%TABLENAME%) is not of type char, varchar, nchar, nvarchar or lvarchar
BTS60	bts error - cannot acquire exclusive lock for %PARAM1%
BTS61	bts error - cannot acquire read lock for %PARAM1%
BTS62	bts error - cannot acquire write lock for %PARAM1%
BTS63	bts error - parameter %PARAM1% is not implemented yet"
BTS64	bts error - %PARAM1% contains a '/' character which indicates an xmlpath however xmlpath_processing is not enabled. Either remove the '/' in the xmltag or enable xmlpath_processing"
BTS65	bts error - invalid value for tempspace parameter: %PARAM1% should be an existing extspace or sbspace
BTS66	bts error - the include_contents cannot be enabled unless the xmltags, all_xmltags, json_names or all_json_names parameter is enabled

SQLstate	Description
BTS67	bts error - include_namespaces cannot be enabled unless either xmltags is specified or all_xmltags is enabled
BTS68	bts error - include_subtag_text cannot be enabled unless either xmltags is specified or all_xmltags is enabled
BTS69	bts error - %PARAM1% only works with on one bts virtual processor
BTS70	bts internal error - mi_lo_specset_sbspace() failed. File %FILE%, line %LINE%
BTS71	bts internal error - mi_lo_stat() failed. File %FILE%, line %LINE%
BTS72	bts internal error - mi_lo_stat_cspec() failed. File %FILE%, line %LINE%
BTS73	bts error - sbspace %PARAM1% is not logged
BTS74	bts error - sbspace for FSE is not set
BTS75	bts error - SBSPACENAME not set in onconfig file
BTS76	bts error - transaction uses too much memory. Perform smaller transactions or increase the value of the xact_memory parameter on the index
BTS77	bts error - invalid value for xact_memory: %PARAM1% should be either unlimited or the maximum amount of memory (between 1 and %PARAM2% kilobytes)
BTS78	bts error - SQL create index and drop index are not supported on updatable secondary nodes
BTS79	bts error - not implemented yet
BTS80	bts error - database must be logged
BTS81	bts error - not in a transaction
BTS82	bts error - xpath syntax error
BTS83	bts internal error - mi_file_seek failed. File %FILE%, line %LINE%
BTS84	bts internal error - mi_lo_decrefcount failed. File %FILE%, line %LINE%
BTS85	bts internal error - mi_lo_from_string failed. File %FILE%, line %LINE%
BTS86	bts internal error - mi_lo_release() failed. File %FILE%, line %LINE%
BTS87	bts internal error - mi_lo_to_string failed. File %FILE%, line %LINE%
BTS88	bts error - no lo handle found in file %PARAM1%
BTS89	bts error - valid lo handle found in file %PARAM1%
BTS90	bts error - CLucene index exists and is locked
BTS91	bts error - CLucene index exists
BTS92	bts error - CLucene index does not exist
BTS93	bts error - the parameter %PARAM1% should be in the form of name="value"
BTS94	bts error - missing a double quotation mark: ". The parameter %PARAM1% should be in the form of name="value"
BTS95	bts error - missing the closing parenthesis:). The parameter %PARAM1% should be in the form of name="(values)"
BTS96	bts error - missing a double quotation mark: ". The parameter %PARAM1% should be in the form of name="(values)"
BTS97	bts error - missing a comma (,) between parameters
BTS98	bts error - duplicate parameters, %PARAM1%, were specified
BTS99	bts clucene error: Unknown error: %PARAM1%
BTSA1	bts clucene error: IO error: %PARAM1%
BTSA2	bts clucene error: Null pointer error: %PARAM1%
BTSA3	bts clucene error: Runtime error: %PARAM1%

SQLstate	Description
BTSA4	bts clucene error: Illegal argument: %PARAM1%
BTSA5	bts clucene error: Parse error: %PARAM1%
BTSA6	bts clucene error: Token manager error: %PARAM1%
BTSA7	bts clucene error: Unsupported operation: %PARAM1%
BTSA8	bts clucene error: Invalid state: %PARAM1%
BTSA9	bts clucene error: Index out of bounds: %PARAM1%
BTSB0	bts clucene error: Too Many Clauses: %PARAM1%
BTSB1	bts clucene error: RAM Transaction error: %PARAM1%
BTSB2	bts clucene error: Invalid Cast: %PARAM1%
BTSC0	GLS Error: An attempt to create a locale with incompatible code sets has occurred
BTSC1	GLS Error: Bad format found in the codeset registry file
BTSC2	GLS Error: Either locale or code set conversion specifiers, i.e., GLS or NLS environment variables, is incorrect, or the codeset name registry file could not be found
BTSC3	GLS Error: Not enough memory to allocate a new locale object or a new codeset conversion object
BTSC4	GLS Error: The locale contains characters that are wider than the library allows
BTSC5	GLS Error: The locale object version is not compatible with the current library
BTSC6	GLS Error: The locale or codeset conversion file could not be found, is not readable, or has the wrong format
BTSC7	GLS Error: Unknown %PARAM1%
BTSC8	bts internal error - mi_lo_stat_size() failed. File %FILE%, line %LINE%
BTSC9	bts internal error - biginttoasc() failed. File %FILE%, line %LINE%
BTSD0	bts error - invalid canonical map[%PARAM1%]: zero length original character string
BTSD1	bts error - invalid canonical map[%PARAM1%]: %PARAM2% is an uppercase character. Uppercase characters are not allowed in canonical maps
BTSD2	bts error - invalid canonical map[%PARAM1%]: missing %PARAM2% in mapped characters specification
BTSD3	bts error - invalid canonical map[%PARAM1%]: missing %PARAM2% in original characters specification
BTSD4	bts error - invalid canonical map[%PARAM1%]: missing : in mapped characters specification
BTSD5	bts error - invalid canonical map[%PARAM1%]: missing] in alternates of original characters specification
BTSD6	bts error - invalid canonical map[%PARAM1%]: spaces found in original character string at %PARAM2%
BTSD7	bts error - invalid canonical map[%PARAM1%]: trailing characters found
BTSD8	bts error - missing the closing parenthesis,), in a string that has an opening parenthesis: (
BTSD9	bts error - missing the column name in table:%PARAM1%. Use the form table:table_name.column_name
BTSE0	bts error - parameter %PARAM1% is not updatable
BTSE1	bts error - unknown parameter name: %PARAM1%
BTSE2	bts error - recursive params parameter
BTSE3	bts error - invalid value for the %PARAM1% parameter: %PARAM2% is too long
BTSE4	bts error - invalid flag for the create_mode parameter: %PARAM1%
BTSE5	bts error - invalid value for the create_mode parameter: %PARAM1% should be a hexadecimal number

SQLstate	Description
BTSE6	bts error - %PARAM1% encoding is not supported for %PARAM2%
BTSE7	bts error - UDR analyzer function %PARAM1% not found
BTSE8	bts error - UDR analyzer function id not found for %PARAM1%
BTSE9	bts error - default analyzer already set
BTSF0	bts error - empty stopwords field specification
BTSF1	bts error - invalid analyzer value: %PARAM1%
BTSF2	bts error - invalid value for the analyzer parameter: %PARAM1%
BTSF3	bts error - no analyzer specified for field: %PARAM1%
BTSF4	bts error - no field name in field:analyzer specification: %PARAM1%
BTSF5	bts error - the field %PARAM1% appears multiple times in the stopwords list
BTSF6	bts error - a stopwords list cannot be specified for the analyzer: %PARAM1%
BTSF7	bts error - too many colons found in stopwords field specification
BTSF8	bts error - too many colons found in field:analyzer specification: %PARAM1%
BTSF9	bts error - there is no snowball stemmer language specified after the period
BTSG0	bts error - there is no snowball stemmer language support for the $DB_LOCALE setting: %PARAM1%
BTSG1	bts error - there is no snowball stemmer language support for the specified language: %PARAM1%
BTSG2	bts error - internal index length %PARAM1% is too long. The maximum is %PARAM2%
BTSG3	bts error - bts_lock_setup: cannot get vp lock pointer
BTSG4	bts error - bts_lock_setup: vp is not locked
BTSG5	bts error - bts_lock_setup: vp is not locked by the current transaction
BTSG6	bts error - not (-) operator may not be specified in thesaurus
BTSG7	bts error - and (+) operator may not be specified in thesaurus
BTSG8	bts error - cannot determine index owner of thesaurus index %PARAM1%
BTSG9	bts error - cannot lock thesaurus index %PARAM1%
BTSH0	bts error - cannot read thesaurus index parameters for %PARAM1%
BTSH1	bts error - the index %PARAM1% does not have the thesaurus parameter set
BTSH2	bts error - thesaurus index cannot be fragmented
BTSH3	bts error - invalid term found in thesaurus. Only word terms should be specified
BTSH4	bts error - the %PARAM1% attribute must be specified
BTSH5	bts error - the text or file attribute must be specified
BTSH6	bts error - the copy_temp attribute can only be specified on an index in an sbspace
BTSH7	bts error - the field is not in the document
BTSH8	bts error - the directory cannot contain a bts index
BTSH9	bts error - the ID is out of bounds
BTSI0	bts error - must be a DBSA to use parameter %PARM1%
BTSI1	bts error - cannot cast json value to bson value: %MSG%
BTSI2	bts error - bson or json types cannot be used in a composite index
BTSI3	bts error - the json_path_processing parameter cannot be enabled unless the json_names parameter is specified or the all_json_names parameter is enabled
BTSI4	bts error - the only_json_values parameter requires that the include_contents is enabled when json_names is specified or all_json_names is enabled

SQLstate	Description
BTSI5	bts error - bson format error decoding type %TYPE% (%SIZE% bytes) at byte %POS% of %MAX% total bytes
BTSI6	bts error - the total number of expected bytes recorded in the bson value, %LEN%, exceeds the actual length of the bson value, %MAX% bytes
BTSI7	bts error - bson format error: %BYTES% bytes of the bson value after the end-of-value mark were not processed
BTSI8	bts error - the json_names, all_json_names, only_json_values, or ignore_json_format_errors parameter can be specified only with bson or json types
BTSI9	bts error - bson or json types cannot be used when xmltags is specified or all_xmltags is enabled
BTSJ0	bts error - json format error at %POS%: extra characters following right brace (})
BTSJ1	bts error - json format error at %POS%: missing right square bracket (])
BTSJ2	bts error - json format error at %POS%: invalid number
BTSJ3	bts error - json format error at %POS%: missing colon (:) separator between name and value
BTSJ4	bts error - json format error at %POS%: missing document
BTSJ5	bts error - json format error at %POS%: missing left brace ({)
BTSJ6	bts error - json format error at %POS%: missing name
BTSJ7	bts error - json format error at %POS%: missing right brace (})
BTSJ8	bts error - json format error at %POS%: missing value
BTSJ9	bts error - json format error at %POS%: missing double quote (") in string

Related reference:

"ignore_json_format_errors index parameter" on page 17-5

Part 5. Hierarchical data type

The node data type helps to resolve a difficult relational database problem–transitive closure.

This transitive closure problem is endemic to data management problems, and not particularly well addressed by the relational model. The same basic problem is found modeling organizational hierarchies, networks, manufacturing and process control databases.

You can use the node data type to improve query performance for many recursive queries. Using the node data type can also ease the burden of transitive dependency in the relational database model. *Transitive dependency* occurs when a non-key attribute is dependent on another non-key attribute. This relationship frequently has multiple levels of attribute dependency. The problem usually is seen when you model organizational hierarchies, networks, and databases for manufacturing and process control.

Chapter 23. The node data type for querying hierarchical data

The node data type is an opaque type of variable length up to 256 characters.

The Scheduler must be running in the database server. If the Scheduler is not running when you create a node data type, a message that the data type is not found is returned.

The database that contains the node data types must meet the following requirements:
- The database must be logged.
- The database must not be defined as an ANSI database.

If you attempt to create a node data type in an unlogged or ANSI database, the message `DataBlade registration failed` is printed in the database server message log.

Operations involving Enterprise Replication are supported.

Deep copy and LIKE matching statements are not supported.

You cannot directly upgrade the unsupported Node DataBlade module on IBM developerWorks® to this version of the node data type.

Troubleshooting the node data type

Error message specific to the node data type have the prefix UND. You can enable tracing on the node data type to diagnose problems.

You might receive the following errors:

Error	Description
UNDE1: Invalid input string.	A node is invalid. Nodes cannot end in 0.
UNDE2: Illegal character found in input string.	An argument contains an illegal character. Nodes can contain only numeric characters.
UNDE3: Third input parameter is not descendant of first input parameter.	The third argument of a Graft function is not a descendant of the first argument.
UNDE4: Index to node element should be greater than or equal to 1.	A problem exists with the node indexing.

To enable tracing, create a trace class by inserting a record into the **systemtraceclasses** system catalog:

```
INSERT INTO informix.systraceclasses(name) VALUES ('Node');
```

For more details regarding tracing, see the *IBM Informix Guide to SQL: Reference*.

Chapter 24. Node data type functions

Use these functions in queries involving the node data type.

Ancestors() function

The **Ancestors()** function is an iterator function that returns ancestor nodes. The Ancestors function recursively calls itself with the output from IsAncestor.

Syntax

```
Ancestors(node)
```

node The node for which you want to find all ancestor nodes.

Example

```
EXECUTE FUNCTION ancestors('1.2.3.4.5.6.7.8.9');
```

This function returns the following eight rows as ancestor nodes:

```
1.2.3.4.5.6.7.8
1.2.3.4.5.6.7
1.2.3.4.5.6
1.2.3.4.5
1.2.3.4
1.2.3
1.2
1.0
```

Compare() function

The **Compare( )** function compares two node types to determine if they are the same.

Returns: -1, 0, or 1.

-1 The first argument is less than the second.

0 The arguments are equal.

1 The first argument is greater than the second.

Syntax

```
compare(node1, node2)
```

node1 The first node to compare.

node2 The node to which the first argument is compared

Example

```
CREATE TABLE nodetab1 (col1 node);
INSERT INTO nodetab1 VALUES ('1.0');
INSERT INTO nodetab1 VALUES ('2.0');

SELECT n1.col1, n2.col1, Compare (n1.col1, n2.col1)
FROM nodetab1 n1, nodetab1 n2;

col1         1.0
col1         1.0
```

```
(expression)  0

col1          2.0
col1          1.0
(expression)  1

col1          1.0
col1          2.0
(expression)  -1
```

Depth() function

The **Depth( )** function returns the number of levels in the specified node.

Returns: integer

Syntax

Depth(*node*)

node The node for which you want to determine depth.

Examples

Example 1

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

Returns: 3

Example 2

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

Returns: 6

Equal() 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.

Examples

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() function

The **GetMember( )** function returns information about a node level, returns integer. The **GetMember()** function returns specific parts of the node argument. The second argument specifies the level you want returned. A NULL is returned if no corresponding level exists.

Returns: integer or NULL

Syntax

GetMember(*node*, *integer*)

node

integer

Example

```
CREATE TABLE nodetab1 (col1 node);
INSERT INTO nodetab1 VALUES ('1.0');
INSERT INTO nodetab1 VALUES ('1.1.1');
INSERT INTO nodetab1 VALUES ('1.1.2');
INSERT INTO nodetab1 VALUES ('1.1.2.1');
INSERT INTO nodetab1 VALUES ('2.0');

SELECT  col1, GetMember(col1, 3)
FROM    nodetab1;

col1          1.0
(expression)

col1          1.1.1
(expression)  1

col1          1.1.2
(expression)  2

col1          1.1.2.1
(expression)  2

col1          2.0
(expression)
```

GetParent() function

The **GetParent( )** function returns the parent of a node. If the node does not have a parent NULL is returned.

Returns: node or NULL

Syntax

GetParent(*node*)

node The child node whose parent you want to determine.

Example

```
CREATE TABLE nodetab1 (col1 node);
INSERT INTO nodetab1 VALUES ('1.0');
INSERT INTO nodetab1 VALUES ('1.1.1');
INSERT INTO nodetab1 VALUES ('1.1.2');
INSERT INTO nodetab1 VALUES ('1.1.2.1');
INSERT INTO nodetab1 VALUES ('2.0');
```

```
SELECT  col1, GetParent(col1)
FROM    nodetab1;

col1         1.0
(expression)

col1         1.1.1
(expression)  1.1

col1         1.1.2
(expression)  1.1

col1         1.1.2.1
(expression)  1.1.2

col1         2.0
(expression)
```

Graft() 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

```
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() 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*.

Examples

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() function

The **GreaterThanOrEqual()** function compares two nodes to determine if the first is greater or equal to the second. Implements the comparison operator and can be used in SQL statements either using the function name or the corresponding symbol.

Returns: Boolean

Syntax

`GreaterThanOrEqual(`*`node1`*`, `*`node2`*`)`

node1 The node that you are will compare against.

node2 The node that you are checking to see if it is greater than or equal to *node1*.

Examples

Example 1

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

Example 2

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

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

Increment() function

The **Increment( )** function determines the next node at the same level. You can also increase the level of a node by one at a specified level.

Returns: node

Syntax

`Increment(`*`node`*`, `*`integer`*`)`

node The starting node to increment from.

integer
: The node member to increment. If you do not specify this argument, the next node at the same level as *node1* is returned.

Examples

Example 1

```
EXECUTE FUNCTION Increment('1.2.3');
(expression)  1.2.4
```

This example uses only one argument. The result shows the next node at the same level.

Example 2

```
EXECUTE FUNCTION Increment('1.2.3', 3);
(expression)  1.2.4
```

This example increments the member in position three, whose value is 3.

Example 3

```
EXECUTE FUNCTION Increment('1.2.3', 1);
(expression)  2.0
```

This example increments the first node member.

IsAncestor() function

The **IsAncestor()** function lets you determine if a specific node is an ancestor of another. This function is the opposite of **IsDescendant()**.

Returns: Boolean

Syntax

`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`.

Examples

Example 1

```
CREATE TABLE nodetab1 (col1 node);
INSERT INTO nodetab1 VALUES ('1.0');
INSERT INTO nodetab1 VALUES ('1.1');
INSERT INTO nodetab1 VALUES ('1.1.1');

SELECT  n1.col1, n2.col1, IsAncestor (n1.col1, n2.col1)
FROM    nodetab1 n1, nodetab1 n2;

col1          1.0
col1          1.1
(expression)  t

col1          1.0
col1          1.1.1
(expression)  t

col1          1.1
```

```
col1          1.1.1
(expression)  t

col1          1.1.1
col1          1.1
(expression)  f
```

Example 2

```
SELECT  col1
FROM    nodetab1 n1
WHERE   isAncestor(col1, '1.1.2');

col1  1.0

col1  1.1
```

IsChild() function

The **IsChild( )** function determines whether a node is a child of another node. This is the opposite of **IsParent()**.

Returns: Boolean

Syntax

`IsChild(`*node1*`,` *node2*`)`

node1 The node that you want to determine whether it is a child of *node2*.

node2 The parent node for which you want to find a child.

Example

```
CREATE TABLE nodetab1 (col1 node);
INSERT INTO nodetab1 VALUES ('1.0');
INSERT INTO nodetab1 VALUES ('1.1');
INSERT INTO nodetab1 VALUES ('1.1.1');

SELECT  n1.col1, n2.col1, IsChild (n1.col1, n2.col1)
FROM    nodetab1 n1, nodetab1 n2;

col1          1.1
col1          1.0
(expression)  t

col1          1.1.1
col1          1.0
(expression)  f

col1          1.0
col1          1.1
(expression)  f

col1          1.1
col1          1.1
(expression)  f

col1          1.1.1
col1          1.1
(expression)  t

col1          1.0
col1          1.1.1
(expression)  f
```

IsDescendant() function

The **IsDescendant( )** function lets you determine if a specific node is a descendant of another. This function is the opposite of **IsAncestor()**.

Returns: Boolean

Syntax

```
IsDescendant(node1, node2)
```

node1 The node that you want to determine whether it is a descendant of `node1`.

node2 The parent node for which you want to find a descendant.

Example

```
CREATE TABLE nodetab1 (col1 node);
INSERT INTO nodetab1 VALUES ('1.0');
INSERT INTO nodetab1 VALUES ('1.1');
INSERT INTO nodetab1 VALUES ('1.1.1');

SELECT  n1.col1, n2.col1, IsDescendant (n1.col1, n2.col1)
FROM    nodetab1 n1, nodetab1 n2;

col1          1.0
col1          1.0
(expression)  f

col1          1.1
col1          1.0
(expression)  t

col1          1.1.1
col1          1.0
(expression)  t

col1          1.0
col1          1.1
(expression)  f
```

IsParent() function

The **IsParent( )** function lets you determine if a specific node is a parent of another. This function is the opposite of **IsChild()**.

Returns: Boolean

Syntax

```
IsParent(node1, node2)
```

node1 The node that you want to determine whether it is a parent of *node2*.

node2 The descendant node for which you want to find a parent.

Example

```
CREATE TABLE nodetab1 (col1 node);
INSERT INTO nodetab1 VALUES ('1.0');
INSERT INTO nodetab1 VALUES ('1.1');
INSERT INTO nodetab1 VALUES ('1.1.1');

SELECT  n1.col1, n2.col1, IsParent (n1.col1, n2.col1)
FROM    nodetab1 n1, nodetab1 n2;
```

```
col1         1.0
col1         1.1
(expression)  t

col1         1.1
col1         1.1.1
(expression)  t

col1         1.0
col1         1.1.1
(expression)  f
```

Length() Node function

The **Length( )** function returns the number of levels in the specified node and is equivalent to the **Depth()** function. This is the name of the function that was included in Node Version 1.0 and supported for continuity.

Returns: integer

Syntax

`Length(node::node)`

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

Example

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

LessThan() function

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

Returns: Boolean

Syntax

`LessThan(node1, node2)`

node1 The node that you are will compare against.

node2 The node that you are checking to see if it is less than *node1*.

Examples

Example 1

```
SELECT * FROM tablename WHERE LessThan(nodecolumn, '1.4');
```

The following list includes nodes that are less than 1.4:

1. 0.4
2. 1.3
3. 1.3.66
4. 1.1.1.1

The following list includes nodes that are greater than 1.4:

1. 1.4.1.1
2. 1.5
3. 2.0

Example 2

```
SELECT * FROM tablename WHERE nodecolumn < '1.4';
```

LessThanOrEqual() function

The **LessThanOrEqual()** function compares two nodes to determine if the first is less or equal to the second. Implements the comparison operator and can be used in SQL statements either using the function name or the corresponding symbol.

Returns: Boolean

Syntax

LessThanOrEqual(*node1*, *node2*)

node1 The node that you are will compare against.

node2 The node that you are checking to see if it is less than or equal to *node1*.

Examples

Example 1

```
SELECT * FROM tablename
WHERE LessThanOrEqual(nodecolumn, '1.4');
```

This example searches the values in the node column of the table to find the node with the value 1.4.

Example 2

```
SELECT * FROM tablename
WHERE nodecolumn <= '1.4';
```

This example is the equivalent to the first, but uses symbols instead of the function name.

NewLevel() function

The **NewLevel( )** function creates a new node level. This function simply returns a new node value under the argument node. This function is independent of table values. The function does not check for duplication.

Returns: node

Syntax

NewLevel(*node*)

node The node under which a new node is created

Example

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

NodeRelease() function

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

Returns: string

Syntax

```
NodeRelease()
```

node

NotEqual() function

The **NotEqual( )** function compares two nodes to determine whether they are not equal. Implements the comparison operator and can be used in SQL statements either using the function name or the corresponding symbol. The opposite function is **Equal()**.

Returns: Boolean

Syntax

NotEqual(*node1*, *node2*)

node1 The node against which you will test for inequality.

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

Examples

Example 1

```
SELECT * FROM tablename WHERE NotEqual(nodecolumn, '1.4');
```

Example 2

```
SELECT * FROM tablename WHERE nodecolumn != '1.4';
```

This example is the same as Example 1, except a not equal sign is used in place of the function name.

Part 6. Informix web feature service for Geospatial Data

You use the IBM Informix web feature service extension to add an Open Geospatial Consortium (OGC) web feature service (WFS) as a presentation layer for Informix spatial data types and geodetic data.

See the Informix or machine notes for details on support of WFS.

An OGC web feature service allows requests for geographical features across the web using platform-independent calls.

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

Chapter 25. Informix web feature service administration

The IBM Informix WFS handles requests for geographical features from a web server using platform-independent calls. The Informix WFS is based on the transaction WFS specification from the Open Geospatial Consortium (OGC).

You can use the Informix spatial data types to support web-based geographical programs using data that you have stored in Informix databases.

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.

The Informix WFS 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 information about whether the Informix WFS runs on your operating system, see the machine notes for your platform.

The Scheduler must be running in the database server. If the Scheduler is not running when you run the **wfsdriver** program, a message that the program is not found is returned.

The database that contains the **wfsdriver** program must meet the following requirements:
- The database must be logged.
- The database must not be defined as an ANSI database.

If you attempt to run the **wfsdriver** program in an unlogged or ANSI database, the message `DataBlade registration failed` is printed in the database server message log.

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 IBM 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.

WFSVP virtual processor class

Informix WFS routines run in a virtual processor class named WFSVP.

A WFSVP virtual processor is created automatically the first time you run a WFS routine. You can increase the number of WFSVP virtual processors

To add WFSVP virtual processors, add the following line to your `onconfig` file, substituting *n* with the number of virtual processors you want to start, and restart the database server: `VPCLASS wfsvp,noyield,num=`*n*.

Configuring the WFSDriver program

Before your web server can run the WFSDriver CGI program, you must set up your environment and configure your web server.

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`.

1. If necessary, install the spatial extension, if you chose not to install it with the database server.
2. Run WFSSetup as described in "WFSSetup program" on page 26-9.
3. Run WFSRegister on the tables on which you want to use the web feature service. See "WFSRegister UDR" on page 26-9 for details.
4. 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.

WFS transactions

The transaction operation includes insert, update, and delete operations on web-accessible feature instances. After a transaction completes, the IBM Informix WFS 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 26-3.

Implement security in WFS

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 26-9 for more information.

Chapter 26. WFS reference

The Informix WFS includes elements, programs, routines, and operations.

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 26-1. 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`

Table 26-1. Keys of GetCapabilities (continued)

Key	Mandatory or Optional	Definition and Example
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 operation

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="QName"/>
<.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 an 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 26-2. 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.

Table 26-2. Actions corresponding to idgen values (continued)

idgen Value	Action
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 an IBM Informix 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. WFS 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 IBM 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 collection or row types in the table definition. Only IBM Informix base types and spatial data types are supported.

Run the **WFSRegister()** UDR on a table before using it with the WFS:

```
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 26-8 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. SQL Packages Extension

The SQL packages extension provides SPL routines that you can use in an application that is compatible with database servers other than Informix.

The SQL packages extension is a built-in extension in the `extend/excompat` directory of your installation. You must manually register the extension using the instructions in the *IBM Informix DataBlade Module Installation and Registration Guide*.

The database that contains the SQL packages extension must meet the following requirements or the extension is not registred:

- The database must be logged.
- The database must not be defined as an ANSI database.

The following modules are included in the SQL packages extension:

- DBMS_ALERT
- DBMS_LOB
- DBMS_OUTPUT
- DBMS_RANDOM
- UTL_FILE

Related reference:

ondblog: Change Logging Mode (Administrator's Reference)

Chapter 27. DBMS_ALERT package

The DBMS_ALERT package provides a set of procedures for registering for alerts, sending alerts, and receiving alerts.

Alerts are stored in the DBMS_ALERT_EVENTS, DBMS_ALERT_REGISTERED, and DBMS_ALERT_SIGNALED tables which are created in your database when you register the package.

The DBMS_ALERT package includes the following system-defined routines.

Table 27-1. System-defined routines available in the DBMS_ALERT package

Routine name	Description
REGISTER procedure	Registers the current session to receive a specified alert.
REMOVE procedure	Removes registration for a specified alert.
REMOVEALL procedure	Removes registration for all alerts.
SIGNAL procedure	Signals the occurrence of a specified alert.
SET_DEFAULTS procedure	Sets the polling interval for the WAITONE and WAITANY procedures.
WAITANY procedure	Waits for any registered alert to occur.
WAITONE procedure	Waits for a specified alert to occur.

Usage notes

The procedures in the DBMS_ALERT package are useful when you want to send an alert for a specific event. For example, you might want to send an alert when a trigger is activated as the result of changes to one or more tables.

REGISTER procedure

The REGISTER procedure registers the current session to receive a specified alert.

Syntax

```
►►──DBMS_ALERT.REGISTER──(──name──)──────────────────────────────────────►◄
```

Procedure parameters

name
An input argument of type VARCHAR(128) that specifies the name of the alert.

REMOVE procedure

The REMOVE procedure removes registration from the current session for a specified alert.

Syntax

```
>>—DBMS_ALERT.REMOVE—(—name—)——————————————————————————————————————><
```

Procedure parameters

name
An input argument of type VARCHAR(128) that specifies the name of the alert.

REMOVEALL procedure

The REMOVEALL procedure removes registration from the current session for all alerts.

Syntax

```
>>—DBMS_ALERT.REMOVEALL————————————————————————————————————————————><
```

SET_DEFAULTS

The SET_DEFAULTS procedure sets the polling interval that is used by the WAITONE and WAITANY procedures.

Syntax

```
>>—DBMS_ALERT.SET_DEFAULTS—(—sensitivity—)—————————————————————————><
```

Procedure parameters

sensitivity
An input argument of type INTEGER that specifies an interval in seconds for the WAITONE and WAITANY procedures to check for signals. If a value is not specified, then the interval is 1 second by default.

SIGNAL procedure

The SIGNAL procedure signals the occurrence of a specified alert. The signal includes a message that is passed with the alert. The message is distributed to the listeners (processes that have registered for the alert) when the SIGNAL call is issued.

Syntax

```
>>—DBMS_ALERT.SIGNAL—(—name—,—message—)————————————————————————————><
```

Procedure parameters

name
An input argument of type VARCHAR(128) that specifies the name of the alert.

message
An input argument of type VARCHAR(32672) that specifies the information to

pass with this alert. This message can be returned by the WAITANY or WAITONE procedures when an alert occurs.

WAITANY procedure

The WAITANY procedure waits for any registered alerts to occur.

Syntax

```
►►—DBMS_ALERT.WAITANY—(—name—,—message—,—status—,—timeout—)—————————►◄
```

Procedure parameters

name
An output argument of type VARCHAR(128) that contains the name of the alert.

message
An output argument of type VARCHAR(32672) that contains the message sent by the SIGNAL procedure.

status
An output argument of type INTEGER that contains the status code returned by the procedure. The following values are possible

0 An alert occurred.

1 A timeout occurred.

timeout
An input argument of type INTEGER that specifies the amount of time in seconds to wait for an alert.

WAITONE procedure

The WAITONE procedure waits for a specified alert to occur.

Syntax

```
►►—DBMS_ALERT.WAITONE—(—name—,—message—,—status—,—timeout—)—————————►◄
```

Procedure parameters

name
An input argument of type VARCHAR(128) that specifies the name of the alert.

message
An output argument of type VARCHAR(32672) that contains the message sent by the SIGNAL procedure.

status
An output argument of type INTEGER that contains the status code returned by the procedure. The following values are possible

0 An alert occurred.

1 A timeout occurred.

timeout
An input argument of type INTEGER that specifies the amount of time in seconds to wait for the specified alert.

Chapter 28. DBMS_LOB package

The DBMS_LOB package provides the capability to operate on large objects.

In the following sections describing the individual procedures and functions, lengths and offsets are measured in bytes if the large objects are BLOBs. Lengths and offsets are measured in characters if the large objects are CLOBs.

The DBMS_LOB package supports LOB data up to 10M bytes.

The DBMS_LOB package includes the following routines.

Table 28-1. System-defined routines available in the DBMS_LOB package

Routine Name	Description
APPEND procedure	Appends one large object to another.
COMPARE function	Compares two large objects.
COPY procedure	Copies one large object to another.
ERASE procedure	Erases a large object.
GETLENGTH function	Gets the length of the large object.
INSTR function	Gets the position of the nth occurrence of a pattern in the large object starting at offset.
READ procedure	Reads a large object.
SUBSTR function	Gets part of a large object.
TRIM procedure	Trims a large object to the specified length.
WRITE procedure	Writes data to a large object.

In partitioned database environments, you will receive an error if you execute any of the following routines inside a WHERE clause of a SELECT statement:

- dbms_lob.compare
- dbms_lob.get_storage_limit
- dbms_lob.get_length
- dbms_lob.instr
- dbms_lob.isopen
- dbms_lob.substr

The following table lists the public variables available in the package.

Table 28-2. DBMS_LOB public variables

Public variables	Data type	Value
lob_readonly	INTEGER	0
lob_readwrite	INTEGER	1

APPEND procedures

The APPEND procedures provide the capability to append one large object to another.

Note: Both large objects must be of the same type.

Syntax

```
►►—APPEND_BLOB—(—dest_lob—,—src_lob—)——————————————————————————►◄
```

```
►►—APPEND_CLOB—(—dest_lob—,—src_lob—)——————————————————————————►◄
```

Parameters

dest_lob
: An input or output argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator for the destination object. Must be the same data type as *src_lob*.

src_lob
: An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator for the source object. Must be the same data type as *dest_lob*.

COMPARE function

The COMPARE function performs an exact byte-by-byte comparison of two large objects for a given length at given offsets.

The function returns:

- Zero if both large objects are exactly the same for the specified length for the specified offsets
- Non-zero if the objects are not the same
- Null if *amount*, *offset_1*, or *offset_2* are less than zero.

The large objects being compared must be the same data type.

Syntax

```
►►—COMPARE—(—lob_1—,—lob_2—————————————————————————————————————►

►—┬——————————————————————————————————————————┬—)———————————————►◄
  └—,—amount—┬——————————————————————————————┬┘
             └—,—offset_1—┬———————————————┬┘
                          └—,—offset_2—┘
```

Parameters

lob_1
: An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the first large object to be compared. Must be the same data type as *lob_2*.

lob_2
: An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the second large object to be compared. Must be the same data type as *lob_1*.

amount
: An optional input argument of type INTEGER. If the data type of the large objects is BLOB, then the comparison is made for *amount* bytes. If the data type of the large objects is CLOB, then the comparison is made for *amount* characters. The default is the maximum size of a large object.

offset_1
: An optional input argument of type INTEGER that specifies the position within the first large object to begin the comparison. The first byte (or character) is offset 1. The default is 1.

offset_2
: An optional input argument of type INTEGER that specifies the position within the second large object to begin the comparison. The first byte (or character) is offset 1. The default is 1.

COPY procedures

The COPY procedures provide the capability to copy one large object to another.

The source and destination large objects must be the same data type.

Syntax

```
►►—COPY_BLOB—(—dest_lob—,—src_lob—,—amount——————————————————————►

►—┬————————————————————————————————————┬—)———————————————————————►◄
  └—,—dest_offset—┬——————————————————┬—┘
                  └—,—src_offset—————┘
```

```
►►—COPY_CLOB—(—dest_lob—,—src_lob—,—amount——————————————————————►

►—┬————————————————————————————————————┬—)———————————————————————►◄
  └—,—dest_offset—┬——————————————————┬—┘
                  └—,—src_offset—————┘
```

Parameters

dest_lob
: An input or output argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to which *src_lob* is to be copied. Must be the same data type as *src_lob*.

src_lob
: An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object from which *dest_lob* is to be copied. Must be the same data type as *dest_lob*.

amount
: An input argument of type INTEGER that specifies the number of bytes or characters of *src_lob* to be copied.

dest_offset
An optional input argument of type INTEGER that specifies the position in the destination large object where writing of the source large object should begin. The first position is offset 1. The default is 1.

src_offset
An optional input argument of type INTEGER that specifies the position in the source large object where copying to the destination large object should begin. The first position is offset 1. The default is 1.

ERASE procedures

The ERASE procedures provide the capability to erase a portion of a large object.

To erase a large object means to replace the specified portion with zero-byte fillers for BLOBs or with spaces for CLOBs. The actual size of the large object is not altered.

Syntax

```
>>--ERASE_BLOB--(--lob_loc--,--amount--+-------------+--)----------------><
                                       '-,--offset---'
```

```
>>--ERASE_CLOB--(--lob_loc--,--amount--+-------------+--)----------------><
                                       '-,--offset---'
```

Parameters

lob_loc
An input or output argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to be erased.

amount
An input or output argument of type INTEGER that specifies the number of bytes or characters to be erased.

offset
An optional input argument of type INTEGER that specifies the position in the large object where erasing is to begin. The first byte or character is at position 1. The default is 1.

GETLENGTH function

The GETLENGTH function returns the length of a large object.

The function returns an INTEGER value that reflects the length of the large object in bytes (for a BLOB) or characters (for a CLOB).

Syntax

```
>>--GETLENGTH--(--lob_loc--)---------------------------------------------><
```

Parameters

lob_loc
: An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object whose length is to be obtained.

INSTR function

The INSTR function returns the location of the nth occurrence of a specified pattern within a large object.

The function returns an INTEGER value of the position within the large object where the pattern appears for the nth time, as specified by *nth*. This value starts from the position given by *offset*.

Syntax

```
>>--INSTR--(--lob_loc--,--pattern--+--------------------------+--)-------------><
                                   '-,--offset--+---------+--'
                                                '-,--nth--'
```

Parameters

lob_loc
: An input argument of type BLOB or CLOB that specifies the large object locator of the large object in which to search for the *pattern*.

pattern
: An input argument of type BLOB(32767) or VARCHAR(32672) that specifies the pattern of bytes or characters to match against the large object.

: *pattern* must be BLOB if *lob_loc* is a BLOB; and *pattern* must be VARCHAR if *lob_loc* is a CLOB.

offset
: An optional input argument of type INTEGER that specifies the position within *lob_loc* to start searching for the *pattern*. The first byte or character is at position 1. The default value is 1.

nth
: An optional argument of type INTEGER that specifies the number of times to search for the *pattern*, starting at the position given by *offset*. The default value is 1.

READ procedures

The READ procedures provide the capability to read a portion of a large object into a buffer.

Syntax

```
>>--READ_BLOB--(--lob_loc--,--amount--,--offset--,--buffer--)------------------><
```

```
>>--READ_CLOB--(--lob_loc--,--amount--,--offset--,--buffer--)------------------><
```

Parameters

lob_loc
: An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to be read.

amount
: An input or output argument of type INTEGER that specifies the number of bytes or characters to read.

offset
: An input argument of type INTEGER that specifies the position to begin reading. The first byte or character is at position 1.

buffer
: An output argument of type BLOB(32762) or VARCHAR(32672) that specifies the variable to receive the large object. If *lob_loc* is a BLOB, then *buffer* must be BLOB. If *lob_loc* is a CLOB, then *buffer* must be VARCHAR.

SUBSTR function

The SUBSTR function provides the capability to return a portion of a large object.

The function returns a BLOB(32767) (for a BLOB) or VARCHAR (for a CLOB) value for the returned portion of the large object read by the function.

Syntax

```
>>—SUBSTR—(—lob_loc—┬——————————————————————————┬—)——————————————————————><
                    └—,—amount—┬——————————————┬—┘
                               └—,—offset—————┘
```

Parameters

lob_loc
: An input argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to be read.

amount
: An optional input argument of type INTEGER that specifies the number of bytes or characters to be returned. The default value is 32,767.

offset
: An optional input argument of type INTEGER that specifies the position within the large object to begin returning data. The first byte or character is at position 1. The default value is 1.

TRIM procedures

The TRIM procedures provide the capability to truncate a large object to the specified length.

Syntax

```
>>—TRIM_BLOB—(—lob_loc—,—newlen—)————————————————————————————————————————><
```

```
►►—TRIM_CLOB—(—lob_loc—,—newlen—)——————————————————►◄
```

Parameters

lob_loc
: An input or output argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to be trimmed.

newlen
: An input argument of type INTEGER that specifies the new number of bytes or characters to which the large object is to be trimmed.

WRITE procedures

The WRITE procedures provide the capability to write data into a large object.

Any existing data in the large object at the specified offset for the given length is overwritten by data given in the buffer.

Syntax

```
►►—WRITE_BLOB—(—lob_loc—,—amount—,—offset—,—buffer—)——————————►◄
```

```
►►—WRITE_CLOB—(—lob_loc—,—amount—,—offset—,—buffer—)——————————►◄
```

Parameters

lob_loc
: An input or output argument of type BLOB(10M) or CLOB(10M) that specifies the large object locator of the large object to be written.

amount
: An input argument of type INTEGER that specifies the number of bytes or characters in *buffer* to be written to the large object.

offset
: An input argument of type INTEGER that specifies the offset in bytes or characters from the beginning of the large object for the write operation to begin. The start value of the large object is 1.

buffer
: An input argument of type BLOB(32767) or VARCHAR(32672) that contains the data to be written to the large object. If *lob_loc* is a BLOB, then *buffer* must be BLOB. If *lob_loc* is a CLOB, then *buffer* must be VARCHAR.

Chapter 29. DBMS_OUTPUT package

The DBMS_OUTPUT package provides a set of procedures for putting messages (lines of text) in a message buffer and getting messages from the message buffer. These procedures are useful during application debugging when you need to write messages to standard output.

The DBMS_OUTPUT package includes the following system-defined routines.

Table 29-1. System-defined routines available in the DBMS_OUTPUT package

Routine name	Description
DISABLE procedure	Disables the message buffer.
ENABLE procedure	Enables the message buffer.
GET_LINE procedure	Gets a line of text from the message buffer.
GET_LINES procedure	Gets one or more lines of text from the message buffer and places the text into a collection.
NEW_LINE procedure	Puts an end-of-line character sequence in the message buffer.
PUT procedure	Puts a string that includes no end-of-line character sequence in the message buffer.
PUT_LINE procedure	Puts a single line that includes an end-of-line character sequence in the message buffer.

Use the command line processor (CLP) command SET SERVEROUTPUT ON to redirect the output to standard output.

DISABLE and ENABLE procedures are not supported inside autonomous procedures.

An autonomous procedure is a procedure that, when called, executes inside a new transaction independent of the original transaction.

DISABLE procedure

The DISABLE procedure disables the message buffer.

After this procedure runs, any messages that are in the message buffer are discarded. Calls to the PUT, PUT_LINE, or NEW_LINE procedures are ignored, and no error is returned to the sender.

Syntax

```
►►──DBMS_OUTPUT.DISABLE──────────────────────────────────────────────►◄
```

ENABLE procedure

The ENABLE procedure enables the message buffer. During a single session, applications can put messages in the message buffer and get messages from the message buffer.

Syntax

```
►►─DBMS_OUTPUT.ENABLE─(─buffer_size─)──────────────────────────────►◄
```

Procedure parameters

buffer_size
: An input argument of type INTEGER that specifies the maximum length of the message buffer in bytes. If you specify a value of less than 2000 for *buffer_size*, the buffer size is set to 2000. If the value is NULL, then the default buffer size is 20000.

GET_LINE procedure

The GET_LINE procedure gets a line of text from the message buffer. The text must be terminated by an end-of-line character sequence.

Syntax

```
►►─DBMS_OUTPUT.GET_LINE─(─line─,─status─)──────────────────────────►◄
```

Procedure parameters

line
: An output argument of type VARCHAR(32672) that returns a line of text from the message buffer.

status
: An output argument of type INTEGER that indicates whether a line was returned from the message buffer:

- 0 indicates that a line was returned
- 1 indicates that there was no line to return

GET_LINES procedure

The GET_LINES procedure gets one or more lines of text from the message buffer and stores the text in a collection. Each line of text must be terminated by an end-of-line character sequence.

Syntax

```
►►─DBMS_OUTPUT.GET_LINES─(─lines─,─numlines─)──────────────────────►◄
```

Procedure parameters

lines
: An output argument of type DBMS_OUTPUT.CHARARR that returns the lines

of text from the message buffer. The type DBMS_OUTPUT.CHARARR is internally defined as a VARCHAR(32672) ARRAY[2147483647] array.

numlines
: An input and output argument of type INTEGER. When used as input, specifies the number of lines to retrieve from the message buffer. When used as output, indicates the actual number of lines that were retrieved from the message buffer. If the output value of *numlines* is less than the input value, then there are no more lines remaining in the message buffer.

NEW_LINE procedure

The NEW_LINE procedure puts an end-of-line character sequence in the message buffer.

Syntax

```
►►──DBMS_OUTPUT.NEW_LINE──────────────────────────────────────────►◄
```

PUT procedure

The PUT procedure puts a string in the message buffer. No end-of-line character sequence is written at the end of the string.

Syntax

```
►►──DBMS_OUTPUT.PUT──(──item──)───────────────────────────────────►◄
```

Procedure parameters

item
: An input argument of type VARCHAR(32672) that specifies the text to write to the message buffer.

PUT_LINE procedure

The PUT_LINE procedure puts a single line that includes an end-of-line character sequence in the message buffer.

Syntax

```
►►──DBMS_OUTPUT.PUT_LINE──(──item──)──────────────────────────────►◄
```

Procedure parameters

item
: An input argument of type VARCHAR(32672) that specifies the text to write to the message buffer.

Chapter 30. DBMS_RANDOM package

The DBMS_RANDOM package provides a mechanism for generating random numbers. Use the INITIALIZE procedure to set the seed value, which is used by the random number generator to generate the numbers.

After enough repetitions, it is possible that some of the generated values will repeat. To reduce the possibility of repeating values, periodically change the seed value by using the SEED procedure.

The DBMS_RANDOM package includes the following system-defined routines and types.

Table 30-1. System-defined routines available in the DBMS_RANDOM package

Routine name	Description
"INITIALIZE procedure"	Initializes the package with the specified integer seed value. Optional.
SEED procedure	Resets the seed with the specified integer value.
RANDOM function	Uses the existing seed value to return a random integer.
TERMINATE procedure	Terminates the package by resetting the seed value to `0`. Optional.

INITIALIZE procedure

The INITIALIZE procedure initializes the system package with the specified integer seed value and is optional.

Syntax

```
►►──DBMS_RANDOM_INITIALIZE ()──────────────────────────────────────────►◄
```

This example:

```
execute procedure dbms_random_initialize (17809465);
```

Returns this output:

```
Routine executed.
```

SEED procedure

The SEED procedure resets the seed with the specified integer value.

Syntax

```
►►──DBMS_RANDOM_SEED ()────────────────────────────────────────────────►◄
```

This example:

```
execute procedure dbms_random_seed (-45902345);
```

Returns this output:

```
Routine executed.
```

RANDOM function

The RANDOM function uses the seed value to return a random integer.

Syntax

```
►►──DBMS_RANDOM_RANDOM ()──────────────────────────────────────────►◄
```

This example:

```
insert into random_test VALUES (0, dbms_random_random());
```

Returns this output:

```
1 row(s) inserted.
```

TERMINATE procedure

The TERMINATE procedure terminates the use of the system package by resetting the seed value to `0` and is optional.

Syntax

```
►►──DBMS_RANDOM_TERMINATE ()───────────────────────────────────────►◄
```

This example:

```
execute procedure dbms_random_terminate ();
```

Returns this output:

```
Routine executed.
```

Chapter 31. UTL_FILE package

The UTL_FILE package provides a set of routines for reading from and writing to files on the database server file system.

The UTL_FILE system package includes the following system-defined routines and types.

Table 31-1. System-defined routines available in the UTL_FILE package

Routine name	Description
FCLOSE procedure	Closes a specified file.
FCLOSE_ALL procedure	Closes all open files.
FFLUSH procedure	Flushes unwritten data to a file.
FOPEN function	Opens a file.
GET_LINE procedure	Gets a line from a file.
NEW_LINE procedure	Writes an end-of-line character sequence to a file.
PUT procedure	Writes a string to a file.

The following list describes all of the named conditions that an application can receive.

Table 31-2. Named conditions for an application

Condition Name	Description
access_denied	Access to the file is denied by the operating system.
charsetmismatch	A file was opened using FOPEN_NCHAR, but later I/O operations used non-CHAR functions such as PUTF or GET_LINE.
delete_failed	Unable to delete file.
file_open	File is already open.
internal_error	Unhandled internal error in the UTL_FILE system package.
invalid_filehandle	File handle does not exist.
invalid_filename	A file with the specified name does not exist in the path.
invalid_maxlinesize	The MAX_LINESIZE value for FOPEN is invalid. It must be 1 - 32672.
invalid_mode	The open_mode argument in FOPEN is invalid.
invalid_offset	The ABSOLUTE_OFFSET argument for FSEEK is invalid. It must be greater than 0 and less than the total number of bytes in the file.
invalid_operation	File could not be opened or operated on as requested.

Table 31-2. Named conditions for an application (continued)

Condition Name	Description
invalid_path	The specified path does not exist or is not visible to the database.
read_error	Unable to read the file.
rename_failed	Unable to rename the file.
write_error	Unable to write to the file.

FCLOSE procedure

The FCLOSE procedure closes a specified file.

Syntax

```
►►──UTL_FILE.FCLOSE──(──file──)──────────────────────────────────────►◄
```

Procedure parameters

file
: An input or output argument of type UTL_FILE.FILE_TYPE that contains the file handle. When the file is closed, this value is set to 0.

FCLOSE_ALL procedure

The FCLOSE_ALL procedure closes all open files. The procedure runs successfully even if there are no open files to close.

Syntax

```
►►──UTL_FILE.FCLOSE_ALL──────────────────────────────────────────────►◄
```

FFLUSH procedure

The FFLUSH procedure forces unwritten data in the write buffer to be written to a file.

Syntax

```
►►──UTL_FILE.FFLUSH──(──file──)──────────────────────────────────────►◄
```

Procedure parameters

file
: An input argument of type UTL_FILE.FILE_TYPE that contains the file handle.

FOPEN function

The FOPEN function opens a file for I/O.

Syntax

```
►►─UTL_FILE.FOPEN─(─location─,─filename─,─open_mode─┬──────────────────┬─)──────►◄
                                                    └─,─max_linesize─┘
```

Return value

This function returns a value of type UTL_FILE.FILE_TYPE that indicates the file handle of the opened file.

Function parameters

location
: An input argument of type VARCHAR(128) that specifies the alias of the directory that contains the file.

filename
: An input argument of type VARCHAR(255) that specifies the name of the file.

open_mode
: An input argument of type VARCHAR(10) that specifies the mode in which the file is opened:

  a Append to file

  r Read from file

  w Write to file

max_linesize
: An optional input argument of type INTEGER that specifies the maximum size of a line in characters. The default value is 1024 bytes. In read mode, an exception is thrown if an attempt is made to read a line that exceeds *max_linesize*. In write and append modes, an exception is thrown if an attempt is made to write a line that exceeds *max_linesize*. End-of-line characters do not count toward the line size.

GET_LINE procedure

The GET_LINE procedure gets a line of text from a specified file. The line of text does not include the end-of-line terminator. When there are no more lines to read, the procedure throws a NO_DATA_FOUND exception.

Syntax

```
►►─UTL_FILE.GET_LINE─(─file─,─buffer─)──────────────────────────────►◄
```

Procedure parameters

file
: An input argument of type UTL_FILE.FILE_TYPE that contains the file handle of the opened file.

buffer
: An output argument of type VARCHAR(32672) that contains a line of text from the file.

NEW_LINE procedure

The NEW_LINE procedure writes an end-of-line character sequence to a specified file.

Syntax

```
►►—UTL_FILE.NEW_LINE—(—file——————————————)——————————————————————————————►◄
                           └—,—lines—┘
```

Procedure parameters

file
An input argument of type UTL_FILE.FILE_TYPE that contains the file handle.

lines
An optional input argument of type INTEGER that specifies the number of end-of-line character sequences to write to the file. The default is 1.

PUT procedure

The PUT procedure writes a string to a specified file. No end-of-line character sequence is written at the end of the string.

Syntax

```
►►—UTL_FILE.PUT—(—file—,—buffer—)——————————————————————————————————————►◄
```

Procedure parameters

file
An input argument of type UTL_FILE.FILE_TYPE that contains the file handle.

buffer
An input argument of type VARCHAR(32672) that specifies the text to write to the file.

Part 8. Appendixes

Appendix. Accessibility

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

Accessibility features for IBM Informix products

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 products. These features support:

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

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.

IBM and accessibility

For more information about the IBM commitment to accessibility, see the *IBM Accessibility Center* at http://www.ibm.com/able.

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 that follows the % symbol is the name of a syntax fragment rather than a literal. For example, the line 2.1 %OP1 refers 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 can write `HOST STATE`, but you cannot 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 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. This material may be available from IBM in other languages. However, you may be required to own a copy of the product or product version in that language in order to access it.

IBM may not offer the products, services, or features discussed in this document in other 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 grant 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:

Intellectual Property Licensing
Legal and Intellectual Property Law
IBM Japan, Ltd.
19-21, Nihonbashi-Hakozakicho, Chuo-ku
Tokyo 103-8510, 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 websites are provided for convenience only and do not in any manner serve as an endorsement of those websites. The materials at those websites are not part of the materials for this IBM product and use of those websites 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 document 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 is for planning purposes only. The information herein is subject to change before the products described become available.

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. The sample programs are provided "AS IS", without warranty of any kind. IBM shall not be liable for any damages arising out of your use of the sample programs.

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.

Privacy policy considerations

IBM Software products, including software as a service solutions, ("Software Offerings") may use cookies or other technologies to collect product usage information, to help improve the end user experience, to tailor interactions with the end user, or for other purposes. In many cases no personally identifiable information is collected by the Software Offerings. Some of our Software Offerings can help enable you to collect personally identifiable information. If this Software Offering uses cookies to collect personally identifiable information, specific information about this offering's use of cookies is set forth below.

This Software Offering does not use cookies or other technologies to collect personally identifiable information.

If the configurations deployed for this Software Offering provide you as customer the ability to collect personally identifiable information from end users via cookies and other technologies, you should seek your own legal advice about any laws applicable to such data collection, including any requirements for notice and consent.

For more information about the use of various technologies, including cookies, for these purposes, see IBM's Privacy Policy at http://www.ibm.com/privacy and IBM's Online Privacy Statement at http://www.ibm.com/privacy/details in the section entitled "Cookies, Web Beacons and Other Technologies", and the "IBM Software Products and Software-as-a-Service Privacy Statement" at http://www.ibm.com/software/info/product-privacy.

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.

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.

Java and all Java-based trademarks and logos are trademarks or registered trademarks of Oracle and/or its affiliates.

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

"informix".mqipolicy table 8-4
"informix".mqipubsub table 8-4
"informix".mqiservice table 8-2

A

Access method
  bts 15-5
Accessibility A-1
  dotted decimal format of syntax diagrams A-1
  keyboard A-1
  shortcut keys A-1
  syntax diagrams, reading in a screen reader A-1
all_json_names index parameter 17-4
all_xmlattrs Basic Text Search index parameter 18-5
all_xmltags Basic Text Search index parameter 18-4
Alnum analyzer 19-4
Analyzers
  basic text searching 19-1
Ancestors() function
  defined 24-1
API interface 3-1
  using 4-6
APPEND procedure 28-2
ASCII representation
  in the Binary DataBlade module 13-1
AUTO_READAHEAD configuration parameter 21-2

B

Basic text search
  all_json_names index parameter 17-4
  Alnum analyzer 19-4
  analyzers 19-1
  building the index in RAM 15-19
  Canonical mapping 15-8
  CJK analyzer 19-4
  composite index 15-13
  creating an index 15-4
  eSoundex analyzer 19-5
  ignore_json_format_errors index parameter 17-5
  include_contents index parameter 17-5
  json_array_processing index parameter 17-6
  json_names index parameter 17-8
  json_path_processing index parameter 17-10
  Keyword analyzer 19-6
  limiting memory use 15-18
  maximum query results 15-12
  maximum tokens 15-12
  only_json_values index parameter 17-11
  Simple analyzer 19-7
  Snowball analyzer 19-8
  Soundex analyzer 19-7
  Standard analyzer 19-10
  Stopword analyzer 19-11
  stopwords 15-15
  thesaurus 15-17
  User-defined analyzer 19-12
  Whitespace analyzer 19-13
Basic Text Search
  Boolean operators 16-9
  boosting a term 16-8
  default sbspace 15-2
  error codes listed 22-1
  escaping special search characters 16-2
  fuzzy searches 16-6
  Grouping words and phrases 16-2
  obtaining a score value 16-1
  overview 0-7
  preparation steps 15-1
  proximity searches 16-7
  query terms 16-2
  range searches 16-8
  requirements 15-1
  restrictions 15-1
  setting SBSPACENAME 15-2
  supported data types 15-1
  transactions 21-1
  wildcard searches 16-5
Basic Text Search DataBlade module
  index 15-5
Basic Text Search fields 16-3
Basic Text Search index parameters
  all_xmlattrs 18-5
  all_xmltags 18-4
  include_contents 18-9
  include_namespaces 18-10
  include_subtag_text 18-12
  strip_xmltags 18-9
  xmlpath_processing 18-7
  xmltags 18-2
Basic text search JSON index parameters
  syntax 17-2
Basic Text Search queries
  restrictions 16-1
Basic Text Search XML index parameters
  overview 18-1
  syntax 18-1
basic text searching
  analyzers 19-1
Basic text searching
  JSON index parameters 17-1
bdtrelease() function 14-3
bdttrace() function 14-4
Binary data
  determining for lld_lob data type 4-2
  indexing 13-3
  inserting 13-2
  inserting into table 4-1
  specifying with lld_lob data type 2-2
Binary data types
  overview 0-3
  restrictions 12-1
Binary DataBlade module
  ASCII representation 13-1
binary18 data type 0-3, 13-1
binaryvar data type 0-3, 13-1
bit_and() function 14-1
bit_complement() function 14-1
bit_or() function 14-2

bit_xor() function 14-2
Bitwise functions 14-1
BLOB data type
  casting to lld_lob data type 2-2
    explicitly 4-2
    implicitly 4-1
Boolean operators
  Basic Text Search 16-9
Boosting a term
  Basic Text Search 16-8
BSON documents
  basic text search index 17-1
bts
  access method 15-5
  operator classes 15-5
  virtual processors 21-2
bts index
  canonical_maps index parameter 15-8
  creating 15-5
  deletion mode 15-11
  directory location 15-3
  optimize 15-11
  restrictions 15-1
bts_blob_ops operator class 15-5
bts_bson_ops operator class 15-5
bts_char_ops operator class 15-5
bts_clob_ops operator class 15-5
bts_contains() search predicate
  syntax 16-1
bts_index_compact() function 20-1
bts_index_fields() function 20-1
bts_json_ops operator class 15-5
bts_longlvarchar_ops operator class 15-5
bts_lvarchar_ops operator class 15-5
bts_release() function 20-4
bts_tracefile() function 20-5
bts_tracelevel() function 20-5
bts_varchar_ops operator class 15-5
BUFFERPOOL configuration parameter 21-2

C

Callback function
  registering 5-1
Canonical mapping 15-8
canonical_maps index parameter 15-8
Casting
  BLOB data type to lld_lob data type 2-2
    explicitly 4-2
    implicitly 4-1
  CLOB data type to lld_lob data type 2-2
    explicitly 4-2
    implicitly 4-1
  lld_lob data type to BLOB and CLOB data types 2-2, 4-1, 4-2
Character data
  determining for lld_lob data type 4-2
  inserting into table 4-1
  specifying with lld_lob data type 2-2
Chinese text search 19-4
CJK analyzer 19-4
Client files
  copying
    to a large object 3-16, 3-19
    to a large object, example 4-1, 4-4, 4-5
  creating 3-14
  deleting 3-15
Client files *(continued)*
  functions 3-14
  opening 3-17
CLOB data type
  casting to lld_lob data type 2-2
    explicitly 4-2
    implicitly 4-1
COMPARE function 28-2
Compare() function
  defined 24-1
compliance with standards xi
Composite index
  basic text searching 15-13
Concurrent access, how to limit 1-2
Configuration parameters
  MQCHLLIB 9-2
  MQSERVER 9-1
Conventions
  functions, naming 3-1
COPY procedure 28-3

D

Data types
  binary18 13-1
  binaryvar 13-1
  lld_lob
    casting to BLOB and CLOB data types 2-2, 4-1, 4-2
    defined 2-2
    determining type of data 3-22, 4-2
    introduced 1-1
    using 4-1, 4-2
    using to insert binary and character data into table 4-1
  lld_locator
    defined 2-1
    introduced 1-1
    using 4-3, 4-5
    using to insert row into table 4-3
    using to reference smart large object, example 4-4
DBMS_LOB module
  APPEND procedures 28-2
  COMPARE function 28-2
  COPY procedures 28-3
  ERASE procedures 28-4
  GETLENGTH function 28-4
  INSTR function 28-5
  READ procedures 28-5
  SUBSTR function 28-6
  TRIM procedures 28-6
  WRITE procedures 28-7
DBMS_LOB package
  overview 28-1
DBMS_OUTPUT package 29-1
DBMS_RANDOM package 30-1
Default table values
  MQ 8-1
Deletion modes
  bts index 15-11
Depth() function
  defined 24-2
Disabilities, visual
  reading syntax diagrams A-1
Disability A-1
Disk space
  for the bts index 21-2
Dotted decimal format of syntax diagrams A-1

E

ENABLE procedure 29-2
Equal() function
  defined 24-2
ERASE procedure 28-4
Error code
  argument 5-1
Error codes
  MQ 10-1
Errors
  callback functions, registering for 5-1
  codes listed 5-2
  codes listed, Basic Text Search 22-1
  error code argument for 5-1
  exceptions, generating for 3-20
  exceptions, handling for 5-1
  handling
    example of 4-6
    functions for 3-20
    MQ 10-1
    SQL 5-1
  status of, and function return value 5-1
  translating to SQL states 3-20
Escaping special search characters
  Basic Text Search 16-2
eSoundex analyzer 19-5
ESQL/C
  interface 3-1
Exceptions
  generating 3-20
  handling 5-1

F

FCLOSE procedure 31-2
FCLOSE_ALL procedure 31-2
FFLUSH procedure 31-2
Fields
  in Basic Text Search 16-3
Files
  client. 3-22
  copying smart large objects to 3-22
  creating, example 4-5
  deleting, example 4-5
FOPEN function 31-3
functions
  FOPEN 31-3
  RANDOM 30-2
Functions
  Ancestors() 24-1
  basic large object 3-2
  bdtrelease() 14-3
  bdttrace() 14-4
  bit_and() 14-1
  bit_complement() 14-1
  bit_or() 14-2
  bit_xor() 14-2
  bitwise 14-1
  bts_index_compact() 20-1
  bts_index_fields() 20-1
  bts_release() 20-4
  bts_tracefile() 20-5
  bts_tracelevel() 20-5
  client file support 3-14
  Compare() 24-1
  Depth() 24-2
Functions *(continued)*
  Equal() 24-2
  error code argument 5-1
  error utility 3-20
  GetMember() 24-3
  GetParent() 24-3
  Graft() 24-4
  GreaterThan() 24-4
  GreaterThanOrEqual() 24-5
  Increment() 24-5
  introduced 1-1
  IsAncestor() 24-6
  IsChild() 24-7
  IsDescendant() 24-8
  IsParent() 24-8
  Length() 24-9
  LENGTH() 14-4
  LessThan() 24-9
  LessThanOrEqual() 24-10
  lld_close() 3-3
    using 4-6
  lld_copy() 3-3
    using 4-4, 4-5
  lld_create 3-5, 4-4
  lld_create_client() 3-14
  lld_delete_client() 3-15
  lld_delete() 3-7
  lld_error_raise() 3-20
  lld_from_client() 3-16
    using 4-4
  LLD_LobType 3-22, 4-2
  lld_open_client 3-17
  lld_open() 3-8
    using 4-6
  lld_read() 3-10, 4-6
  lld_sqlstate 3-20
  lld_tell() 3-12
  lld_to_client() 3-19, 4-5
  lld_write() 3-13, 4-6
  LOCopy 3-21
  LOToFile 3-22
  MQCreateVtiRead() 8-8
  MQCreateVtiReceive() 8-10
  MQCreateVtiWrite() 8-12
  MQHasMessage() 8-13
  MQInquire() 8-14
  MQPublish() 8-15
  MQPublishClob() 8-19
  MQRead() 8-23
  MQReadClob() 8-25
  MQReceive() 8-28
  MQReceiveClob() 8-30
  MQSend() 8-32
  MQSendClob() 8-35
  MQSubscribe() 8-37
  MQTrace() 8-39
  MQUnsubscribe() 8-41
  MQVersion() 8-42
  naming conventions 3-1
  NewLevel() 24-10
  NodeRelease() 24-11
  NotEqual() 24-11
  OCTET_LENGTH() 14-4
  return value and error status 5-1
  smart large object copy 3-21
Fuzzy searches
  Basic Text Search 16-6

G

GET_LINE procedure 29-2
  files 31-3
GET_LINES procedure 29-2
GETLENGTH function 28-4
GetMember() function
  defined 24-3
GetParent() function
  defined 24-3
Graft() function
  defined 24-4
GreaterThan() function
  defined 24-4
GreaterThanOrEqual() function
  defined 24-5
Grouping words and phrases
  Basic Text Search 16-2

H

Hexadecimal representation
  in the Binary DataBlade module 13-1

I

ignore_json_format_errors index parameter 17-5
include_contents Basic Text Search index parameter 18-9
include_contents index parameter 17-5
include_namespaces Basic Text Search index parameter 18-10
include_subtag_text Basic Text Search index parameter 18-12
Increment() function
  defined 24-5
Indexing binary data 13-3
industry standards xi
Informix
  configuring for MQ 6-2
INITIALIZE procedure 30-1
Inserting binary data 13-2
INSTR function 28-5
Interfaces 3-1
  API 3-1
    using 4-6
  ESQL/C 3-1
  naming conventions 3-1
  SQL 3-2
    using 4-1, 4-5
IsAncestor() function
  defined 24-6
IsChild() function
  defined 24-7
IsDescendant() function
  defined 24-8
IsParent() function
  defined 24-8

J

Japanaese text search 19-4
JSON documents
  basic text search index 17-1
JSON index parameters 17-1
  syntax for basic text search 17-2
json_array_processing index parameter 17-6
json_names index parameter 17-8
json_path_processing index parameter 17-10

K

Keyword analyzer 19-6
Korean text search 19-4

L

Large Object Locator 0-1
  functions 1-1
Large objects
  accessing 0-1
  basic functions for 3-2
  closing 3-3
  copying
    client files to 3-16
    function for 3-3
    to client files 3-19
    to large objects, example 4-4
  copying to client files, example 4-5
  creating 3-5
  defined 0-1
  deleting 3-7
  limiting concurrent access 1-2
  offset
    returning for 3-12
  opening 3-8
  protocols, listed 2-1
  reading from 3-10
  referencing 2-1
  setting read and write position in 3-10
  tracking open 5-1
  writing to 3-13
Length() function
  defined 24-9
LENGTH() function 14-4
LessThan() function
  defined 24-9
LessThanOrEqual() function
  defined 24-10
Libraries
  API 3-1
  ESQL/C 3-1
  SQL 3-2
lld_close() function 3-3
  using 4-6
lld_copy() function 3-3
  using 4-4, 4-5
lld_create_client() function 3-14
lld_create() function 3-5
  using 4-4
lld_delete_client() function 3-15
lld_delete() function 3-7
lld_error_raise() function 3-20
lld_from_client() function 3-16
  using 4-4
lld_lob data type
  casting to BLOB and CLOB data types 2-2, 4-1
    explicitly 4-2
  defined 2-2
  determining type of data in 3-22, 4-2
  inserting binary data into table 4-1
  inserting character data into table 4-1
  introduced 1-1
  using 4-1, 4-2
LLD_LobType function 3-22
  using 4-2

lld_locator data type
defined 2-1
inserting a row into a table 4-3
introduced 1-1
referencing a smart large object 4-4
using 4-3, 4-5
lld_open_client() function 3-17
lld_open() function 3-8
using 4-6
lld_read() function 3-10
using 4-6
lld_sqlstate() function 3-20
lld_tell() function 3-12
lld_to_client() function 3-19
using 4-5
lld_write() function 3-13
using 4-6
LOCopy function 3-21
LOToFile function 3-22

M

Messages
receiving from a queue 6-2
sending to a queue 6-2
Messaging
WMQ 0-3
MQ
configuration parameters 9-1
configuring 6-2
configuring the server for 6-3, 11-1
default table values 8-1
error codes 10-1
error handling 10-1
functions 6-1
binding a table 7-1
creating a table 7-1
retrieving a queue element 7-2
installing WMQ 6-2
MQ
communications 6-1
preparing 6-1
publishing to queue 6-6
subscribing to queue 6-6
tables 6-1
verifying functionality 6-5
MQ DataBlade
functions
MQCreateVtiRead() 8-8
MQCreateVtiReceive() 8-10
MQCreateVtiWrite() 8-12
MQHasMessage() 8-13
MQInquire() 8-14
MQPublish() 8-15
MQPublishClob() 8-19
MQRead() 8-23
MQReadClob() 8-25
MQReceive() 8-28
MQReceiveClob() 8-30
MQSend() 8-32
MQSendClob() 8-35
MQSubscribe() 8-37
MQTrace() 8-39
MQUnsubscribe() 8-41
MQVersion() 8-42
overview 8-1
inserting data into queue 6-5
MQ DataBlade *(continued)*
publishing to queue 6-7
reading entry from queue 6-5
receiving entry from queue 6-6
unsubscribing from queue 6-7
MQ messaging
server based 6-4
switching between server and client 6-4
mq virtual processor class 6-2
MQCHLLIB configuration parameter 9-2
MQCHLTAB configuration parameter 9-2
MQCreateVtiRead() function
defined 8-8
MQCreateVtiReceive() function
defined 8-10
MQCreateVtiWrite() function
defined 8-12
MQHasMessage() function
defined 8-13
MQInquire() function
defined 8-14
mqm group 6-2
MQPublish() function
defined 8-15
MQPublishClob() function
defined 8-19
MQRead() function
defined 8-23
MQReadClob() function
defined 8-25
MQReceive() function
defined 8-28
MQReceiveClob() function
defined 8-30
MQSend() function
defined 8-32
MQSendClob() function
defined 8-35
MQSERVER configuration parameter 9-1
MQSubscribe() function
defined 8-37
MQTrace() function
defined 8-39
MQUnsubscribe() function
defined 8-41
MQVersion() function
defined 8-42

N

Name service cache 9-1
Naming conventions 3-1
NEW_LINE procedure 29-3, 31-4
NewLevel() function
defined 24-10
Node data type
functions
Ancestors() 24-1
Compare() 24-1
Depth() 24-2
Equal() 24-2
GetMember() 24-3
GetParent() 24-3
Graft() 24-4
GreaterThan() 24-4
GreaterThanOrEqual() 24-5
Increment() 24-5

Node data type *(continued)*
functions *(continued)*
IsAncestor() 24-6
IsChild() 24-7
IsDescendant() 24-8
IsParent() 24-8
Length() 24-9
LessThan() 24-9
LessThanOrEqual() 24-10
NewLevel() 24-10
NodeRelease() 24-11
NotEqual() 24-11
NodeRelease() function
defined 24-11
NotEqual() function
defined 24-11

O

Obtaining a score value
Basic Text Search 16-1
OCTET_LENGTH() function 14-4
Offset
in large objects
returning 3-12
only_json_values index parameter 17-11
Operator classes
for bts 15-5
Optimizing
bts index 15-11

P

package
DBMS_LOB 28-1
DBMS_RANDOM 30-1
UTL_FILE 31-1
packages
DBMS_OUTPUT 29-1
procedures
ENABLE 29-2
FCLOSE 31-2
FCLOSE_ALL 31-2
FFLUSH 31-2
GET_LINE 29-2, 31-3
GET_LINES 29-2
INITIALIZE 30-1
NEW_LINE 29-3, 31-4
PUT 29-3, 31-4
PUT_LINE 29-3
SEED 30-1
SET_DEFAULTS 27-2
SIGNAL 27-2
TERMINATE 30-2
WAITANY 27-3
WAITONE 27-3
Protocol
list, for large objects 2-1
Proximity searches
Basic Text Search 16-7
PUT procedure
put partial line in message buffer 29-3
write string to file 31-4
PUT_LINE procedure
put complete line in message buffer 29-3

Q

Queries
Basic Text Search 16-1
Query results, maximum number 15-12
Query syntax
Basic Text Search 16-1
Query terms
Basic Text Search 16-2

R

RANDOM function 30-2
Range searches
Basic Text Search 16-8
READ COMMITTED
with Basic Text Search 21-1
READ procedure 28-5
Requirements
Basic Text Search 15-1
RESIDENT configuration parameter 21-2
Resources
cleaning up 1-2
Restrictions
Basic Text Search 15-1
Basic Text Search queries 16-1
bts index 15-1
Rollback
limits on with Large Object Locator 1-2

S

sbspace
for bts index 15-3
SBSPACENAME configuration parameter
setting for Basic Text Search 15-2
Schema mapping to WMQ objects 7-1
Score value
Basic Text Search 16-1
Screen reader
reading syntax diagrams A-1
Search predicate
bts_contains() 16-1
Secondary access method
bts 15-5
SEED procedure 30-1
SET_DEFAULTS procedure 27-2
Shortcut keys
keyboard A-1
SIGNAL procedure 27-2
Simple analyzer 19-7
Smart large objects
copying to a file 3-22
copying to a smart large object 3-21
creating, example 4-4
functions for copying 3-21
referencing with lld_lob data type 2-2
referencing, example 4-1
Snowball analyzer 19-8
Soundex analyzer 19-7
SQL
errors 5-1
interface 3-2
using 4-1, 4-5
states, translating from error codes 3-20
Standard analyzer 19-10
standards xi

Stopword analyzer 19-11
stopwords 15-15
strip_xmltags Basic Text Search index parameter 18-9
SUBSTR scalar function
  details 28-6
Supported data types
  Basic Text Search 15-1
Syntax
  bts_contains() 16-1
  for basic text search JSON index parameters 17-2
  for Basic Text Search XML index parameters 18-1
Syntax diagrams
  reading in a screen reader A-1

T

Table values
  default
    MQ 8-1
TERMINATE procedure 30-2
thesaurus 15-17
Tokens
  maximum indexed 15-12
Transaction rollback
  limits on with Large Object Locator 1-2
Transactions
  with Basic Text Search 21-1
TRIM procedure 28-6
Types. 1-1

U

User-defined analyzer 19-12
User-defined routines
  calling API functions from 3-1
  example 4-6, 4-9
UTL_FILE package 31-1

V

Virtual-Table Interface
  accessing WMQ queues 7-1
Visual disabilities
  reading syntax diagrams A-1
VP
  bts 21-2
VPCLASS configuration parameter 21-2
VTI
  accessing WMQ queues 7-1

W

WAITANY procedure 27-3
WAITONE procedure 27-3
Whitespace analyzer 19-13
Wildcard searches
  Basic Text Search 16-5
WMQ
  messages
    SELECT 7-2
  messaging 0-3
  metadata table behavior 7-1
  objects
    schema mapping to 7-1
  product documentation 6-2
WMQ *(continued)*
  queues
    accessing 7-1
    configuring 6-2
    INSERTitems into 7-2
    mapping to tables 7-1
  setting up 6-2, 6-3, 11-1
  tables mapped to
    generating errors 7-3
WRITE procedure 28-7

X

XML index parameters
  syntax for Basic Text Search 18-1
xmlpath_processing Basic Text Search index parameter 18-7
xmltags Basic Text Search index parameter 18-2

Printed in USA

SC27-4512-02

Spine information:

Informix Product Family Informix **Version 12.10** **IBM Informix Database Extensions User's Guide**