LOB Locator DataBlade Module

Programmer's Guide

Version 1.2
March 1997
000-3847

ACKNOWLEDGMENTS

Documentation Team: Michael Kline, Rosemary Wakeman, Kim Bostrom, Bob Nowacki

Contributors: Craig Freedman, Bill Madill, Richard Staehli, Tim Brazil

Table of Contents

Introduction

This chapter introduces the *LOB Locator DataBlade Module Programmer's Guide*. Read this chapter for an overview of the information provided in this manual and for an understanding of the conventions used throughout this manual.

About This Guide

This guide explains how to use the LOB Locator™ DataBlade® module to consistently manage large objects outside the database and large object data within the database.

Organization of This Guide

The *LOB Locator DataBlade Module Programmer's Guide* includes the following chapters:

- This Introduction provides an overview of the contents of the guide, describes documentation conventions used, and lists additional books to supplement the information in the *LOB Locator DataBlade Module Programmer's Guide*.
- Chapter 1, "About LOB Locator," explains the basic concepts and operations of the LOB Locator DataBade module.
- Chapter 2, "Data Types," describes the LOB Locator data types.
- Chapter 3, "Functions," provides a complete reference to the LOB Locator functions.
- Chapter 4, "Sample Code," provides sample code that explains how to use the LOB Locator functions and data types.
- Chapter 5, "Error Handling," describes how to handle errors that can be returned when using the LOB Locator functions.

Types of Users

LOB Locator is intended for database administrators who want to add LOB Locator functionality to their databases and for developers who want to add LOB Locator functionality to their applications.

Conventions

This section describes the conventions used in this guide. By becoming familiar with these conventions, you can more easily gather information from this guide.

The following conventions are discussed:

- Typographical conventions
- Comment icon conventions

Typographical Conventions

The *LOB Locator DataBlade Module Programmer's Guide* uses a standard set of typographical conventions to introduce new terms, illustrate screen displays, and so forth. The following typographical conventions are used throughout this guide.

Convention	Meaning
italics	Within text, new terms and emphasized words are shown in italics. In command descriptions, variables that you must replace with actual data are also shown in italics.
boldface	Identifiers, filenames, database names, menu items, command names, and other similar terms are shown in boldface.
`monospace`	Information that programs display, information that you enter, and program code examples are printed in a monospace typeface.
KEYWORD	All keywords appear in uppercase characters.

Comment Icons

Throughout this guide, comment icons identify three types of information, as described in the following table.

Icon	Description
	The *warning* icon identifies vital instructions, cautions, or critical information.
	The *important* icon identifies significant information about the feature or operation that is being described.
	The *tip* icon identifies additional details or shortcuts for the functionality that is being described.

The information in these paragraphs is always displayed in italic text.

Additional Documentation

The LOB Locator documentation set includes a printed manual and on-line material.

This section describes the following parts of the documentation set:

- Printed documentation
- On-line documentation
- Related reading

Printed Documentation

The *LOB Locator DataBlade Module Programmer's Guide* is a printed manual that describes LOB Locator, providing an overview to its use and a complete reference to its data types and functions. This manual also contains examples and error messages.

On-Line Documentation

Two types of on-line documentation are available:

- On-line manuals
- Release notes, documentation notes, and machine notes

On-Line Manuals

The *LOB Locator DataBlade Module Programmer's Guide* is provided on a CD so that you can view and search for information on-line. For searches, you can specify a word or phrase and specify which manuals you want to search. You can also place electronic annotations and bookmarks on pages that are of particular interest to you. Pages you view and print from the on-line manual have the same layout and design as the printed manual.

Release Notes, Documentation Notes, Machine Notes

In addition to the Informix set of manuals, the following on-line files, located in the **$INFORMIXDIR/release** directory, supplement the information in this manual.

On-Line File	Purpose
Documentation notes	Describes features that are not covered in the manuals or that have been modified since publication.
Release notes	Describes feature differences from earlier versions of Informix products and how these differences might affect current products. This file also contains information about any known problems and their workarounds.

Please examine these files because they contain vital information about application and performance issues.

Related Reading

For additional information on large objects, consult the following books:

- *Informix Guide to SQL: Reference*
- *Informix Guide to SQL: Syntax*
- *Informix Guide to SQL: Tutorial*

For information on server functions, consult the following book:

- *DataBlade API Programmer's Manual*

For information on using the ESQL / C interface, consult the following book:

- *INFORMIX-ESQL/C Programmer's Manual*

For information about developing DataBlade modules, consult the following documents:

- *DataBlade Developers Kit User's Guide*
- *DataBlade Developers Kit Release Notes*

Informix Welcomes Your Comments

Please tell us what you like or dislike about our manuals. To help us with future versions of our manuals, we want to know about any corrections or clarifications that you would find useful. Please include the following information:

- The name and version of the manual that you are using
- Any comments that you have about the manual
- Your name, address, and phone number

Write to us at the following address:

Informix Software, Inc.
Technical Publications
300 Lakeside Dr., Suite 2700
Oakland, CA 94612

If you prefer to send electronic mail, our address is:

doc@informix.com

Or, send a facsimile to Technical Publications at:

415-926-6571

We appreciate your feedback.

Chapter

1

About LOB Locator

This chapter provides an overview to the LOB Locator DataBade module.

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

In INFORMIX-Universal Server, large object data (data that exceeds a length of 255 bytes or contains non-ASCII characters) is stored 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 *Informix Guide to SQL: Syntax* and *Informix Guide to SQL: Tutorial* for more information.

With LOB Locator you create a reference to a large object and store the reference as a row in the database. The object itself can reside outside the database, for example on a file system (or it could be a BLOB or CLOB type column in the database). The reference identifies the type, or access protocol, of the object and points to its storage location. For example, you could identify an object as a file and provide a pathname to it or identify it as a binary or character smart large object stored in the database. Smart large objects are a category of large objects, including BLOB and CLOB, that store text and images, 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 LOB 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 LOB Locator to reference any type of large object in any storage location. In practice, access protocols must be built into LOB 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 manual—not the manual itself—to see the types of large objects LOB Locator currently supports.*

Using LOB Locator

LOB Locator is implemented through two data types and a set of functions. The next two subsections introduce the data types and functions.

LOB Locator Data Types

LOB Locator defines two data types, lld_locator, and lld_lob.

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

You can also pass an lld_locator row to various LOB 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 LOB Locator function. See “lld_locator” on page 2-3 for a detailed description of this data type.

The lld_lob type enables LOB 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, in addition to pointing to the data, it tracks whether the underlying smart large object contains binary or character data.

See “lld_lob” on page 2-5 for a complete description of this data type.

LOB Locator Functions

LOB 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 LOB Locator functions can be divided into four main categories:

- **Basic functions** for creating, opening, closing, deleting, and reading from and writing to large objects.
- **Client functions** for creating, opening, and deleting client files and for copying large objects to and from client files. After you open a client file, you can use the basic functions to read from and write to the file.
- **Utility functions** for raising errors and converting errors to their SQL state equivalents.
- **Smart large object functions** for copying smart large objects to files and to other smart large objects.

There are three interfaces to the LOB Locator functions:

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

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

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

A limited set of the LOB Locator functions are implemented as user-defined routines that you can execute within SQL statements. See "SQL Interface" on page 3-4 for a list of the LOB Locator functions that you can execute directly in SQL statements.

Chapter 3, "Functions", describes all the LOB Locator functions and the three interfaces in detail.

Limitations

Certain limitations are inherent in using large objects with a database, because the objects themselves, except for smart large objects, are not stored in the database and are not subject to direct control by the server. Two specific areas of concern are transaction rollback and concurrency control.

Transaction Rollback

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

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

Concurrent Access

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

Installation and Registration

LOB Locator is distributed with INFORMIX-Universal Server. To use the LOB Locator functions, you must use BladeManager to register the functions and data types with each database for which you want LOB Locator functionality. See the *BladeManager User's Guide* for more information. This guide also contains some information about installing DataBlade modules.

Chapter

2

Data Types

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

lld_locator

The lld_locator data type identifies a large object. It specifies the kind of large object and provides a pointer to its location. The 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_lob
    lo_location         informix.lvarchar
    )
```

lo_protocol identifies the kind of large object.

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

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

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

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

The *lo_pointer* field uses the lld_lob data type, which is defined by LOB Locator. This data type allows you to point to a smart large object and specify whether it is of type BLOB or type CLOB. See the description of lld_lob on page 2-5 for more information.

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

Figure 2-1 lists the current protocols and summarizes the values for the other fields based on the protocol that you specify. Be sure to check the release notes shipped with this manual to see if LOB 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 LOB Locator all have an IFX prefix.*

Figure 2-1
Fields of lld_locator Data Type

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

Important: *The lo_protocol field is case insensitive. It is shown in uppercase letters for display purposes only.*

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

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

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

You can pass an lld_locator type to these LOB Locator functions:

lld_lob

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

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

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

It is defined for ESQL/C as follows:

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

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

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

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

See "Using the lld_lob Type" on page 4-3 for sample code that uses that features the lld_lob type.

LOB Locator provides explicit casts from:

- a CLOB type to an lld_lob.
- a BLOB type to an lld_lob.
- an lld_lob 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, LOB Locator returns an error message.*

You can pass an lld_lob type to these functions:

- **LOCopy**, page 3-44
- **LOToFile**, page 3-46
- **LLD_LobType**, page 3-47

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

Chapter

3

Functions

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

Interfaces

LOB 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 called out.

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 manual uses the API and ESQL/C naming convention unless referring specifically to an SQL function.

API Library

All LOB 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 LOB Locator API functions from your own user-defined routines. You execute LOB Locator API functions just as you do functions provided by Informix DataBlade API. See the *DataBlade API Programmer's Manual* for more information.

See "Using the API" on page 4-12 for an example of a user-defined routine that calls LOB Locator API functions to copy part of a large object to another large object.

ESQL/C Library

All LOB 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 LOB Locator functions avoid server interaction by directly accessing the underlying large object.

See the *INFORMIX-ESQL/C Programmer's Manual* for more information on using the ESQL / C interface to execute LOB Locator functions.

SQL Interface

The following LOB 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 Informix SQL interface:

- *Informix Guide to SQL: Reference*

- *Informix Guide to SQL: Syntax*
- *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 LOB Locator functions to manipulate data in the open object (see Step 3).

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

3. After you open a large object, you can pass the **LLD_IO** structure to:
   - **lld_tell()**, to return the current position within the large object.
   - **lld_seek()** to change the current position within the object.
   - **lld_read()** to read from large object.
   - **lld_write()** to write to the large object.
   - **lld_close()** to close an object. You should close a large object if the transaction in which you open it is aborted or committed.

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

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

lld_close()

This function closes the specified large object.

Syntax

API

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

ESQL/C

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

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

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

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

Usage

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

Returns

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.

Related Topics

lld_open(), page 3-17

lld_copy()

This function copies the specified large object.

Syntax

API

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

ESQL/C

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

SQL

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

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

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

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

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

Usage

This function copies an existing large object.

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

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

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

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

Specify NULL for the *lo_pointer* field.

Point to the location of the new object in the *lo_location* field.

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

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

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

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

Likewise, if protocols are added to LOB Locator for new types of large objects, these objects might require creation attributes or parameters for which LOB 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.

Returns

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`.

Related Topics

lld_from_client(), page 3-34

lld_to_client(), page 3-40

lld_create()

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

Syntax

API

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

ESQL/C

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

SQL

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

conn	is the connection descriptor established by a previous call to the **mi_open()** or **mi_server_connect()** functions. This parameter is for the API interface only. In the ESQL/C and SQL versions of this function, you must already be connected to a server.
lob	is a pointer to an lld_locator row identifying the object to create.
error	is an output parameter in which the function returns an error code. The SQL version of this function does not have an *error* parameter.

Usage

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

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

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

Specify NULL for the *lo_pointer* field.

Point to the location of the new object in the *lo_location* field.

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

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

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

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

See "Transaction Rollback" on page 1-6 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 LOB Locator for new types of large objects, these objects might require creation attributes or parameters for which LOB 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.

Returns

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`.

Related Topics

lld_delete, page 3-15

lld_open(), page 3-17

lld_delete

This function deletes the specified large object.

Syntax

API

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

ESQL/C

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

SQL

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

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

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

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

Usage

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

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

Returns

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

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

lld_open()

This function opens the specified large object.

Syntax

API

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

ESQL/C

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

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

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

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

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

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

LLD_RDWR	opens the large object for both reading and writing.
LLD_TRUNC	clears the contents of the large object after opening.
LLD_APPEND	seeks to the end of the large object for writing. When the object is opened, the file pointer is positioned at the beginning of the object. If you have opened the object for reading, or reading and writing, you can seek anywhere in the file and read. However, any time you call **lld_write()** to write to the object, the pointer moves to the end of the object to guarantee that you do not overwrite any data.
LLD_SEQ	opens the large object for sequential access only. You cannot use the **lld_seek()** function with the specified large object when this flag is set.
error	is an output parameter in which the function returns an error code.

Usage

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

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

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

See "Limitations" on page 1-6 for more information about transaction rollback and concurrency control.

Returns

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 LOB 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`.

Related Topics

lld_close(), page 3-7

lld_create(), page 3-12

lld_read(), page 3-20

lld_seek(), page 3-22

lld_tell(), page 3-25

lld_write(),page 3-27

lld_read()

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

Syntax

API

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

ESQL/C

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

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

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

bytes is the number of bytes to read.

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

Usage

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

Returns

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`.

Related Topics

lld_open(), page 3-17

lld_seek(), page 3-22

lld_tell(), page 3-25

lld_seek()

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

Syntax

API

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

ESQL/C

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

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

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

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

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

whence determines how the offset is interpreted.

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

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

Usage

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

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

Returns

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.

Related Topics

lld_open(), page 3-17

lld_read(), page 3-20

lld_tell(), page 3-25

lld_write(), page 3-27

lld_tell()

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

Syntax

API

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

ESQL/C

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

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

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

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

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

Usage

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

Returns

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.

Related Topics

lld_open(), page 3-17

lld_read(), page 3-20

lld_seek(), page 3-22

lld_write(), page 3-27

lld_write()

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

Syntax

API

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

ESQL/C

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

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

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

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

bytes is the number of bytes to write.

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

Usage

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

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

Returns

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`.

Related Topics

lld_open(), page 3-17

lld_seek(), page 3-22

lld_tell(), page 3-25

Client File Support

This section describes the LOB 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 simplify the coding of 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 LOB 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 LOB Locator client functions with care. You can only access client files if you are using the client machine on which the files are stored. If you change client machines, you can no longer access files stored on the original client machine. Thus, an application that stores client filenames in the database might find at a later date that the files are inaccessible.

lld_create_client()

This function creates a new client file.

Syntax

API

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

ESQL/C

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

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

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

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

Usage

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

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

See "Transaction Rollback" on page 1-6 for more information.

Returns

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.

Related Topics

lld_delete_client(), page 3-32

lld_delete_client()

This function deletes the specified client file.

Syntax

API

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

ESQL/C

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

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

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

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

Usage

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

Returns

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.

Related Topics

lld_create_client(), page 3-30

lld_from_client()

This function copies a client file to a large object.

Syntax

API

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

ESQL/C

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

SQL

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

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

src is a pointer to the source pathname.

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

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

Usage

This function copies an existing large object.

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

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

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

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

Specify NULL for the *lo_pointer* field.

Point to the location of the new object in the *lo_location* field.

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

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

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

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

Likewise, if protocols are added to LOB Locator for new types of large objects, these objects might require creation attributes or parameters for which LOB 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.

Returns

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`.

Related Topics

lld_create_client(), page 3-30

lld_open_client(), page 3-37

lld_open_client()

This function opens a client file.

Syntax

API

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

ESQL/C

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

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

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

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

LLD_RDONLY	opens the client file for reading only. You cannot use the **lld_write** function to write to the specified client file when this flag is set.
LLD_WRONLY	opens the client file for writing only. You cannot use the **lld_read()** function to read from the specified client file when this flag is set.
LLD_RDWR	opens the client file for both reading and writing.
LLD_TRUNC	clears the contents of the client file after opening.

LLD_APPEND	seeks to the end of the large object for writing. When the object is opened, the file pointer is positioned at the beginning of the object. If you have opened the object for reading, or reading and writing, you can seek anywhere in the file and read. However, any time you call **lld_write()** to write to the object, the pointer moves to the end of the object to guarantee that you do not overwrite any data.
LLD_SEQ	opens the client file for sequential access only. You cannot use the **lld_seek()** function with the specified client file when this flag is set.
error	is an output parameter in which the function returns an error code.

Usage

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

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

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

See "Limitations" on page 1-6 for more information about transaction rollback and concurrency control.

Returns

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 LOB 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`.

Related Topics

lld_close(), page 3-7

lld_read(), page 3-20

lld_seek(), page 3-22

lld_tell(), page 3-25

lld_write(),page 3-27

lld_create_client(), page 3-30

lld_to_client()

This function copies a large object to a client file.

Syntax

API

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

ESQL/C

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

SQL

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

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

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

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

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

Usage

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

Returns

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.

Related Topics

lld_open_client(), page 3-37

Error Utility Functions

The two functions described in this section allow you to:

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

lld_error_raise()

This function generates an exception for the specified error.

Syntax

API

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

error is an error code that you specify.

Usage

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

Returns

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

On failure, it also returns `MI_ERROR`.

lld_sqlstate()

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

Syntax

API

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

ESQL/C

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

error is an error code.

Returns

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

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

Smart Large Object Functions

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

LOCopy

This function creates a copy of a smart large object.

Syntax

SQL

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

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

;
```

lob is a pointer to the smart large object to copy.

table_name is a table name. This parameter is optional.

column_name is a column name. This parameter is optional.

Usage

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

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

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

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

Returns

This function returns a pointer to the new lld_lob value.

Related Topics

LOCopy in the *Informix Guide to SQL: Syntax* manual

LOToFile

Copies a smart large object to a file.

Syntax

SQL

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

lob	is a pointer to the smart large object.
pathname	is a directory path and name of the file to create.
file_dest	is the computer on which the file resides. Specify either `server` or `client`.

Usage

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

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

Returns

This function returns the value of the new filename.

Related Topics

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

LLD_LobType

Returns the type of data in an lld_lob column.

Syntax

SQL

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

lob is a pointer to the smart large object

Usage

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

Returns

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

Chapter

4

Sample Code

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

Using the SQL Interface

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

Using the lld_lob Type

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

Using Implicit lld_lob Casts

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

Figure 4-1
Implicit lld_lob Casts

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

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

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

select * from slobs;


key  1
slo  blob:00608460a6b7c8d900000002000000030000000200000018000000000001000000608
     460736c6f000010029a2a6c92070000000000006c000af0cdd90000008000608250af0c9d
     e

key  2
slo  clob:00608460a6b7c8d900000002000000030000000300000019000000000001000000608
     460736c6f000010029a2a6c930d0000000000006c000af0cdd90000001600000010af0c9d
     e
```

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

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

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

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

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

Using Explicit lld_lob Casts

This example shows how to select large objects of type BLOB and CLOB from a table and how to copy them to a file.

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

Figure 4-2
Explicit lld_lob Casts

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


(expression)  <SBlob Data>


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


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

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

Using the LLD_LobType Function

This section 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 4-4. That example created the table and inserted a BLOB type large object for key 1 and a CLOB type large object for key 2.

Figure 4-3
Using LLD_LobType Function

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

         key (expression)

           1 blob
           2 clob


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


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

The first SELECT statement returns:

```
1 blob
2 clob
```

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

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

Using the lld_locator Type

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

lo_protocol identifies the kind of large object.

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

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

The examples in this section show how to:

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

Inserting an lld_locator Row into a Table

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

Figure 4-4
Inserting an lld_locator Row Into a Table

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

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

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

Creating a Smart Large Object

This example creates a smart large object containing CLOB type data. The **lld_create** function in Figure 4-5 creates a smart large object. The first parameter to **lld_create** uses the IFX_CLOB protocol to specify CLOB as the type of object to create. The other two arguments are NULL.

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

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

Figure 4-5
Using lld_create

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

Copying a Client File to a Large Object

This example uses the **lobs** table created in Figure 4-5 in the previous section.

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

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

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


(expression)  ROW('IFX_CLOB          ','clob:ffffffffa6b7c8d90000000020000000300
              0000090000001a000000000001000000000000ad3c3dc000000000b06eec8000
              00000005c4e6000607fdc00000000000000000000000000',NULL)


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


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

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

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

Copying a Large Object to a Large Object

This example uses the **lobs** table created in Figure 4-4.

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

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

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


(expression)  ROW('IFX_CLOB          ','clob:ffffffffa6b7c8d9000000020000000300
              0000090000001a000000000001000000000000ad3c3dc000000000b06eec8000
              00000005c4e6000607fdc00000000000000000000000000',NULL)


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


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

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

Copying Large Object Data to a Client File

This example uses the **lobs** table created in Figure 4-4. The **lld_toclient** function in Figure 4-8 copies large object data to the **output.txt** client file. This function returns `t` when the function succeeds. The SELECT statement returns `t`, or `true`, indicating that the function returned successfully.

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

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

(expression)

           t
```

Creating and Deleting a Server File

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

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

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

Figure 4-9
Creating and Deleting a Server File

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

select lo from lobs where key = 3;


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


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

(expression)

           t


delete from lobs where key = 3;
```

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

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

Using the API

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

Creating the lld_copy_subset Function

Figure 4-10 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.

Figure 4-10
The lld_copy_subset Function

```
/* 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;
    }
```

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 LOB Locator functions are called for the source object:

- **lld_open** to open the source object
- **lld_seek** to seek to the specified byte offset in the object
- **lld_read** to read the specified number of bytes from the object
- **lld_close** to close the object

The following LOB 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 LOB 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

Although this sample code, in the interest of simplicity and clarity, does not establish a callback for exceptions, you should do so. See the *DataBlade API Programmer's Manual* for more information.

Using the lld_copy_subset Routine

This section shows how to use the **lld_copy_subset** user-defined routine defined in the previous section.

Figure 4-11
Using the lld_copy_subset Routine

```
-- Using the lld_copy_subset function

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

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

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

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

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

Chapter

5

Error Handling

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

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

- Through the error argument of a LOB Locator function
- Through an exception

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

Handling LOB Locator Errors

All LOB 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`.

LOB Locator functions also provide an error code argument that you can test for specific errors. lists the LOB Locator 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, LOB 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 LOB Locator function (API version) can register a callback function. If this function catches and handles an exception and returns control to the LOB Locator function, LOB Locator returns the LLD_E_EXCEPTION error. You can handle this error as you would any other: close open objects and delete objects not inserted in a table.

Handling Exceptions

You should register a callback function to catch exceptions generated by underlying DataBlade API functions called by LOB Locator functions. For example, if you call **lld_read()** to open a smart large object, LOB 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 LOB Locator error codes.

Error Code	SQL State	Description
LLD_E_INTERNAL	ULLD0	Internal LOB Locator error. If you receive this error, call Informix Technical Support.
LLD_E_OK	N.A.	No error.
LLD_E_EXCEPTION	N.A.	MI_EXCEPTION raised and handled. Applies to API only.

(1 of 2)

Error Code	SQL State	Description
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_lob type into a BLOB or CLOB type.
N.A.	ULLDR	Used an invalid import file specification with the lld_lob type.

(2 of 2)

Index

N

O

P

R

S

T

U