IBM ILOG DB Link V5.3

User's Manual

June 2009

Copyright notice

Trademarks

IBM, the IBM logo, ibm.com, Websphere, ILOG, the ILOG design, and CPLEX 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, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

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

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

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

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

Notices

For further information see <*installdir*>/licenses/notices.txt in the installed product.

Table of Contents

About This Manual

This manual tells you how to use the IBM® ILOG® DB Link libraries. More specifically, it explains how to use classes and functions, and includes considerations about particular relational database management systems (RDBMSs). Numerous examples are supplied to help you configure IBM ILOG DB Link depending on your specific RDBMS.

What Is IBM ILOG DB Link?

IBM® ILOG® DB Link gives you a simple yet powerful interface to one or more RDBMSs. Its API (Application Programming Interface) is independent of both the platform and the RDBMS. This applies at several levels:

- **With regard to the RDBMS APIs:** The IBM ILOG DB Link API hides all the proprietary, RDBMS-specific API calls, as well as RDBMS-specific call sequencing. Using one single call, you can have your SQL statement executed.
- **With regard to platforms and compilers:** Your code can be used to communicate with different RDBMSs simultaneously. As IBM ILOG DB Link is platform-independent, you do not have to rewrite any code when changing your platform or compiler. The IBM ILOG DB Link API is similar to the Call Level Interface (CLI) ISO standard but adapted to the C++ language.

- **With regard to the contents of the SQL statements:** Usually, IBM ILOG DB Link does not change the contents of the SQL statements you send to the RDBMS. The only exception is for MS SQL Server, which does not support placeholders. The IBM ILOG DB Link Library interface lacks the necessary functionality. In fact, most of the time, IBM ILOG DB Link does not even read the SQL statements you send. This allows you to send any query you want, but it also means that IBM ILOG DB Link does not check whether your SQL code complies with a standard such as SQL92 or the SQL implementation specific to a given RDBMS.

IBM ILOG DB Link accepts and processes queries that use RDBMS-specific features, but such queries may cause an error with RDBMSs that lack the specific feature.

There are two ways of using IBM ILOG DB Link:

- If the target RDBMSs are known from the beginning, the application can be linked with the appropriate drivers.
- If the application does not specify one or more target RDBMSs, it can be linked with the IBM ILOG DB Link driver manager, which will dynamically load the appropriate drivers when necessary.

Supported RDBMSs and Platforms

The number of RDBMSs and platforms on which IBM® ILOG® DB Link works has been increased. However, not all possible combinations are available. Make sure that your platform, compiler, archtecture and RDBMS versions match a combination marked **Y** for “yes” in Table 1.

If the cell corresponding to your combination does not exist in the table, or if that cell contains a dash or an **N**, you should contact your IBM sales representative.

When a port is obsolete, the cell contains an **O**. In that case, you are strongly advised to switch to another port as soon as possible because future versions of IBM ILOG DB Link will drop that port.

Table 1 indicates which systems and compilers are available with a given RDBMS.

Table 1 *Available Configurations for IBM ILOG DB Link*

IBM® ILOG® Portname	System	Compiler	IBM® DB2®	IBM® Informix®	MS SQL Server *	ODBC	OLE DB	Oracle®	Sybase®
	(Default architecture is 32 bits)		8.x & 9.x	5 to 11	2000, 2005	3.5x	2.x	9 to 11	11,12, 15
ultrasparc32_8_6.2	Solaris 2.8	Forte 6.2	O	O	-	-	-	O	O
ultrasparc64_8_6.2	Solaris 2.8 64 bits	Forte 6.2	Y	Y	-	-	-	Y	Y
ultrasparc32_10_11	Solaris 2.10	SunStudio 11	Y	Y	-	-	-	Y	Y
ultrasparc64_10_11	Solaris 2.8 64 bits	SunStudio 11	Y	Y	-	-	-	Y	Y
alpha_5.1_6.5	Compaq Tru64 V5.1	CXX 6.5	N	O	-	-	-	Y	Y
x86_RHEL4.0_3.4	Red Hat 4.0	gcc 3.4	Y	Y	-	-	-	Y	Y
x86-64_RHEL4.0_3.4	Red Hat Linux 4.0 64 bits	gcc 3.4	Y	Y	-	-	-	Y	Y
x86_sles10.0_4.1	Suze Linux 10.0	gcc 4.1	Y	Y	-	-	-	Y	Y
x86_.net2003_7.1	MS Visual Studio .net2003	msvc 7.1	O	O	O	O	O	O	O
x86_.net2005_8.0	MS Visual Studio .net2005	msvc 8.0	Y	Y	Y	Y	Y	Y	Y
x64_.net2005_8.0	MS Visual Studio .net2005 64 bits	msvc 8.0	Y	Y	Y	Y	Y	Y	Y
x86_.net2008_9.0**	MS Visual Studio 2008	msvc 9.0	Y	Y	Y	Y	Y	Y	Y
x64_.net2008_9.0***	MS Visual Studio 2008 64 bits	msvc 9.0	Y	Y	Y	Y	Y	Y	Y
rs6000_5.1_6.0	AIX 5.1	Visual Age C++ 6.0	Y	Y	-	-	-	Y	Y
power32_5.2_7.0	AIX 5.2 PPC	IBM XL C/C++ 7.0	Y	Y	-	-	-	Y	Y
power64_5.2_7.0	AIX 5.2 PPC 64 bits	IBM XL C/C++ 7.0	Y	Y	-	-	-	Y	Y

Table 1 *Available Configurations for IBM ILOG DB Link (Continued)*

IBM® ILOG® Portname	System	Compiler	IBM® DB2®	IBM® Informix®	MS SQL Server *	ODBC	OLE DB	Oracle®	Sybase®
ia64_hpux11_6.17	Itanium HP UX 11.23 64 bits	aC++ 6.17	-	Y	-	-	-	Y	-
ia32_hpux11_6.17	Itanium HP UX 11.23		-	Y	-	-	-	Y	-
x64_solaris10_11	x86-64 Sun Solaris 5.10	Sun Studio 11		Y	-	-	-	Y	-
x86_solaris10_11	x86-64 Sun Solaris10	Sun Studio 11		Y	-	-	-	Y	-
hp32_11_3.73	HP-UX 11	aCC 3.73	Y	Y	-	-	-	Y	Y
hp64_11_3.73	HP-UX 11 64 bits	aCC 3.73	Y	Y	-	-	-	Y	Y

*Support for MS SQL native DBLib will be discontinued soon. Users are advised to switch to OLE DB port.
**This port was previously known as `x86_windows_vs2008.`
***This port was previously known `as x64_windows_vs2008.`

What You Need to Know

This manual assumes that you are familiar with the operating system in which you are going to use your product. Since this product is written for C++ developers, this manual also assumes that you can write C++ code and that you have a working knowledge of your coding environment. To work with IBM ILOG DB Link, you also need to know SQL.

Manual Organization

This manual is divided as follows:

- *Data Types* provides the correspondence between IBM ILOG DB Link types and RDBMSs, first in output mode, then in input mode.
- *Configuration Issues* describes the environment variables required by IBM ILOG DB Link for each RDBMS, as well as configuration-file issues and some basic configuration features. This chapter also describes what implementation information items can be retrieved from the server.

- *Sessions & Connections* describes how to use the class `IldDbms`.
- *Cursors* describes how to use the class `IldRequest`.
- *Queries* explains how to prepare and execute queries, bind application memory to the database API, access the column descriptors, and retrieve data.
- *Errors and Warnings* explains the error-handling mechanism implemented by IBM ILOG DB Link.
- *Compiling and Linking* discusses compilation flags and compatibility issues, and presents the IBM ILOG DB Link libraries.
- *Code Samples* describes the sample files that are shipped with your IBM ILOG DB Link product.

Notation

The following typographic conventions apply throughout this manual:

- Code extracts and file names are written in `courier` typeface.
- Important ideas are emphasized like *this*.

Naming Conventions

Throughout this manual, we will refer to the "application" as a program that you have written to make use of data in one or more databases, managed by one or more RDBMSs, all linked by IBM ILOG DB Link.

- Basic type names begin with `Il` as they come from the common basic IBM ILOG library (`ilog.lib` or `libilog.a`).
- The names of classes, and functions defined in the IBM ILOG DB Link library begin with `Ild`.
- Constants, error codes, and options are written in uppercase letters, separated by an underscore "_" if their name consists of more than one word:

  ```
  ILD_BAD_FILE
  ```

- The names of classes, functions, C++ types, and enumerated values (`enum`) are written as concatenated, capitalized words:

  ```
  class IldDbms;

  enum IldEntityType {IldTableEntity, IldViewEntity, IldADTEntity,
  IldCallableEntity, IldSynonymEntity};

  IldDbms* IldNewDbms(const char*, const char*);
  ```

- The first word in names of arguments, instances, and member functions begins with a lowercase letter. Other words in such a name begin with an uppercase letter. Data members of classes and fields in structures are prefixed by an underscore "_":

```
 IldRequest::getByteValue();

typedef struct {
   IlInt   _size;
   IldByte* _value;
}
```

- Accessors begin with the keyword `get` followed by the name of the data member:

```
const char* getCursorName() const;
```

- Accessors for Boolean members begin with `is` followed by the name of the data member:

```
IlBoolean isConnected() const;
```

- Modifiers begin with the keyword `set` followed by the name of the data member:

```
IldRequest& setCursorName (const char* cursName);
```

Where to Get More Information

This section tells you where you can find additional information about your product:

- *Related Documentation* lists the other printed and online manuals that make up the IBM® ILOG® DB Link documentation kit.
- *Further Reading* is a short bibliography on the SQL language and RDBMSs.

Related Documentation

In addition to this manual, the IBM ILOG DB Link libraries come with the following documentation:

- The *Reference Manual* describes the various classes and functions in alphabetical order.
- *Release Notes* contain important, last-minute information such as new features and errata, that could not be included in the printed or HTML manuals.
- The `readme.html` file delivered in the standard distribution. This file contains the most current information about platform prerequisites for IBM ILOG DB Link.
- Source code for examples delivered in the standard distribution.

Further Reading

SQL Language

- *"Information Technology - Database Languages - SQL, Part 2: Foundation (SQL/Foundation),"*

  See also the other parts of the standard at `http://www.iso.org/iso/iso_catalogue.htm`.

- *"An Introduction to Database Systems"* 7th edition, C.J. Date, Addison-Wesley, ISBN 0-201-38590-2, August 1999.

- *"A Guide to the SQL Standard"* 4th edition, C.J.Date and H. Darwen, Addison-Wesley, ISBN 0-201-96426-0, 1997.

- *"Database Systems: the Complete Book"*, 1st edition, H. Garcia-Molina, J. D. Ullman, and J. D. Widom, Prentice Hall, ISBN 0-130319-95-3, October 2001.

- `www.sql.org`

RDBMSs

- **DB2**
  - *"SQL Reference"*, IBM, DB2 Documentation, S10J-8165-01
  - "*Call Level Interface Guide and Reference*", IBM, DB2 Documentation, S10J-8159-00
- **Informix**
  - *"Informix-ESQL/C Programmer's Manual"*, Informix Press, Part Number 000-7629
  - "*Informix Answers Online*" Version 1.6, CD, Part No. 000-6823
- **ODBC**
  - *"ODBC 3.0 Programmer's Reference and SDK Guide"*, Microsoft® Press, Part Number 097-0001688
- **OLEDB**
  - *""OLE DB 2.0 Programmer's Reference and Data Access SDK"*, Microsoft® Press, ISBN 0-7356-0590-4
- **Oracle®**
  - *"Programmer's Guide to the Oracle Call Interface"*, Oracle, Part Number 5411-70-1292
  - *"Oracle7 Server SQL Language Reference Manual"*, Oracle, Part Number 778-70-1292

- "*Oracle Call Interface*" Release 8.0 Programmer's Guide 2 Vol. Part Numbers A54657-01 & A54655-01

◆ **Sybase**

- *"Open Client Client Library/C Reference Manual"*, Sybase, Part Number 32840-01-1000-04
- "*Open Client Client Library/C Programmer's Guide"*, Sybase, Part Number 35570-01-1000-03

C H A P T E R

1

Data Types

Individual IBM® ILOG® DB Link type definitions are provided in the *Reference Manual.*

The minimal set of ANSI database data types are translated to the same types for all supported RDBMSs. The data types that are handled by IBM ILOG DB Link are the same for input and for output. The database system input and output types are mapped to IBM ILOG DB Link types according to tables that show which IBM ILOG DB Link type is used to retrieve the database data.

A one-to-one correspondence between database data types and IBM ILOG DB Link types cannot be established. The IBM ILOG DB Link principle for mapping is economy—that is, IBM ILOG DB Link defines a minimal set of data types to which database types are converted. This does not prevent IBM ILOG DB Link from converting all database data types for each supported RDBMS.

This chapter is divided as follows:

- *Output Mode* provides the correspondence between IBM ILOG DB Link types and RDBMSs in output mode.
- *Input Mode* provides the correspondence between IBM ILOG DB Link types and RDBMSs in input mode.

Output Mode

IBM® ILOG® DB Link uses its own types, mapped to C or C++ types, structures, and objects, to fetch or send the data values from or to a database server. Due to variations in the way different RDBMSs implement their own types, the correspondence may vary. However, the SQL standard types, when they exist, are handled by the same IBM ILOG DB Link types, regardless of the RDBMS. That is:

- `CHAR`, `VARCHAR`, `NCHAR`, `LVARCHAR`, and `NVARCHAR` are mapped to `IldStringType`.
- `INTEGER` and `SMALLINT` are mapped to `IldIntegerType`.
- `FLOAT`, `REAL`, and `DOUBLE PRECISION` are mapped to `IldRealType`.
- `NUMBER`, `NUMERIC`, `DECIMAL` are mapped to:
  - `IldRealType` when the default settings are active.
  - `IldStringType` when the "*numeric as string*" feature is turned on.
  - `IlNumeric` when the "*numeric as object*" feature is turned on.
- `DATE`, `TIME`, and `TIMESTAMP` are mapped to `IldDateType` or `IldDateTimeType` when the "*date as string*" feature is turned off.

The following tables are organized by RDBMS. They show which IBM ILOG DB Link type is used to retrieve data from the database. Tables for the following RDBMSs are provided:

- *DB2*
- *Informix*
- *MS SQL Server*
- *ODBC*
- *Oracle*
- *Sybase*

DB2

Table 1.1 *Mapping between IBM ILOG DB Link Types and DB2 Types*

IBM ILOG DB Link Type	SQL Type
`IldByteType`	-
`IldStringType`	CHAR, CHAR FOR BIT DATA VARCHAR, VARCHAR FOR BIT DATA
`IldDateType` `IldDateTimeType`	DATE TIME TIMESTAMP
`IldRealType` [1] `IldNumericType`	DECIMAL NUMERIC FLOAT DOUBLE REAL
`IldIntegerType`	INTEGER, SMALLINT
`IldLongTextType`	LONG VARCHAR
`IldBinaryType`	LONG VARCHAR FOR BIT DATA
`IldBLOBType`	BLOB
`IldCLOBType`	CLOB
`IldDecFloatType`	DEC_FLOAT

[1] When the *numeric as string* feature is turned on, the column data types are converted into `IldStringType`.

Informix

Table 1.2 *Mapping between IBM ILOG DB Link Types and Informix Types*

IBM ILOG DB Link Type	SQL Type
`IldByteType`	-
`IldStringType`	CHAR, CHARACTER NCHAR CHARACTER VARYING, VARCHAR NVARCHAR, LVARCHAR

Table 1.2 *Mapping between IBM ILOG DB Link Types and Informix Types*

IBM ILOG DB Link Type	SQL Type
`IldDateType` `IldDateTimeType`	DATE DATETIME INTERVAL
`IldRealType` [1] `IldNumericType`	DEC, DECIMAL, NUMERIC REAL, SMALLFLOAT DOUBLE PRECISION, FLOAT
`IldIntegerType`	INT, INTEGER SMALLINT SERIAL
`IldMoneyType`(1)	MONEY
`IldLongTextType`	TEXT
`IldBinaryType`	BYTE
`IldCollectionType` [2]	LIST, SET, MULTISET
`IldObjectType`(2)	[NAMED] ROW
`IldCLOBType`	CLOB
`IldBLOBType`	BLOB

[1] When the *numeric as string* feature is turned on, the column data types are converted into `IldStringType`.

[2] Only supported for Informix Universal Server.

MS SQL Server

Table 1.3 *Mapping Between IBM ILOG DB Link Types and MS SQL Server Types*

IBM ILOG DB Link Type	SQL Type
`IldByteType`	TINYINT BIT
`IldIntegerType`	SMALLINT INT
`IldRealType`	NUMERIC [1] DECIMAL(1) FLOAT, DOUBLE PRECISION REAL
`IldMoneyType`	SMALLMONEY MONEY
`IldDateType` `IldDateTimeType`	SMALLDATETIME DATETIME
`IldStringType`	CHAR, NCHAR VARCHAR, NVARCHAR BINARY
`IldLongTextType`	TEXT
`IldBinaryType`	IMAGE

[1] When the *numeric as string* feature is turned on, these column data types are converted into `IldStringType`.

ODBC

Table 1.4 *Mapping Between IBM ILOG DB Link Types and ODBC Types*

IBM ILOG DB Link Type	SQL Type
`IldByteType`	SQL_BIT SQL_TINYINT
`IldIntegerType`	SQL_SMALLINT SQL_INTEGER SQL_BIGINT

Table 1.4 *Mapping Between IBM ILOG DB Link Types and ODBC Types (Continued)*

IBM ILOG DB Link Type	SQL Type
`IldRealType`	SQL_FLOAT SQL_DOUBLE SQL_REAL SQL_DECIMAL [1] SQL_NUMERIC(1)
`IldStringType`	SQL_CHAR SQL_VARCHAR SQL_BINARY SQL_VARBINARY
`IldDateType` `IldDateTimeType`	SQL_DATE SQL_TIME SQL_TIMESTAMP
`IldMoneyType`	[2]
`IldLongTextType`	SQL_LONGVARCHAR
`IldBinaryType`	SQL_LONGVARBINARY

[1] When the *numeric as string* feature is turned on, these column data types are converted into `IldStringType`.

[2] Database Money Type is translated to `NUMERIC` or `DECIMAL`.

Oracle

Table 1.5 *Mapping Between IBM ILOG DB Link Types and Oracle Types*

IBM ILOG DB Link Type	SQL Type
`IldByteType`	-
`IldStringType`	CHAR, CHARACTER VARCHAR, VARCHAR2, CHARACTER VARYING, CHAR VARYING ROWID MLSLABEL RAW
`IldIntegerType` [1] `IldRealType` [2] `IldNumericType`	NUMBER, NUMERIC, DECIMAL, DEC, INTEGER, INT, SMALLINT, FLOAT, DOUBLE PRECISION, REAL, BINARY_FLOAT, BINARY_DOUBLE
`IldDateType` [3] `IldDateTimeType`	DATE,TIMESTAMP, TIMESTAMP WITH TIME ZONE, TIMESTAMP WITH LOCAL TIME ZONE, INTERVAL YEAR TO MONTH, INTERVAL DAY TO SECOND
`IldMoneyType`	-
`IldLongTextType`	LONG, LONGVARCHAR
`IldBinaryType`	LONG RAW
`IldCollectionType`	VARRAY NESTED TABLE
`IldObjectType`	OBJECT
`IldCursorType`	CURSOR
`IldBLOBType`	BLOB
`IldCLOBType`	CLOB

[1] All numeric types are described through the external type SQLT_NUM. DB Link differentiates between integer and floating-point numbers using the precision and scale values. If the scale is non-null or is null and precision is null or greater than 10, or if both scale and precision are null, then the default type is `IldRealType`. Otherwise, it is `IldIntegerType`.
Such a protocol leaves a potential problem for numbers with a precision set to 10 and no scale. They can overflow the C or C++ limit for integer values. If such an overflow occurs due to the values stored in the database system, your application can use the "*numeric as string*" or "*numeric as object*" features.

[2] When the *numeric as string* feature is turned on, these column data types are converted into `IldStringType`.

[3] When the *date as string* feature is turned off, these column data types are converted into `IldDateTimeType`.

Sybase

Table 1.6 *Mapping Between IBM ILOG DB Link Types and Sybase Types*

IBM ILOG DB Link Type	SQL Type
`IldByteType`	TINYINT BIT
`IldIntegerType`	SMALLINT INT
`IldRealType`	NUMERIC [1] DECIMAL(1) FLOAT, DOUBLE PRECISION REAL
`IldMoneyType`	SMALLMONEY MONEY
`IldDateType` `IldDateTimeType`	SMALLDATETIME DATETIME
`IldStringType`	CHAR, NCHAR VARCHAR, NVARCHAR BINARY
`IldLongTextType`	TEXT
`IldBinaryType`	IMAGE

[1] When the *numeric as string* feature is turned on, these column data types are converted into `IldStringType`.

Special Features

The following special features of the Output Mode are described:

- *Date As String*
- *Numeric As String*
- *Numeric As Object*

Date As String

When the "*date as string*" feature is turned off, IBM ILOG DB Link automatically sets the column type to `IldDateTimeType`. As a consequence, it refuses to return the `DATE`, `DATETIME`, or `TIMESTAMP` value in string form and raises the error `ILD_TYPE_MISMATCH`.

See the functions `IldIldBase::useStringDate` and `IldIldBase::setStringDateUse`, for the `IldRequest` and `IldDbms` classes in the *IBM ILOG DB Link Reference Manual* for more information. (Functions common to those classes are documented in the `IldIldBase` class.)

Numeric As String

When the *numeric as string* feature is turned on, IBM ILOG DB Link retrieves the numeric-type value in string form. If the member function `IldRequest::getColRealValue` is inadvertently used to retrieve the column value, the result is unpredictable.

See the functions `IldIldBase::useStringNumeric` and `IldIldBase::setStringNumericUse`, for the `IldRequest` and `IldDbms` classes in the *IBM ILOG DB Link Reference Manual* for more information. (Functions common to those classes are documented in the `IldIldBase` class.)

Numeric As Object

When the *numeric as object* feature is turned on, IBM ILOG DB Link silently retrieves the numeric-type value in `IlNumeric` object form. If the function `IldRequest::getColRealValue` is inadvertently used to retrieve the column value, the result is unpredictable. The various features for numeric-type values retrieval are mutually exclusive.

See the functions `IldIldBase::useNumeric` and `IldIldBase::setNumericUse`, for the `IldRequest` and `IldDbms` classes in the *IBM ILOG DB Link Reference Manual* for more information. (Functions common to those classes are documented in the `IldIldBase` class.)

Input Mode

The following table lists the RDBMS API type-names that IBM® ILOG® DB Link uses to send parameter values to the RDBMSs.

Table 1.7 *RDBMS API Type Symbols Used by IBM ILOG DB Link*

IBM ILOG DB Link	DB2	Informix	MS SQL Server	ODBC
IldByteType	SQL_C_TINYINT	CINTTYPE	SQLINT1	SQL_C_TINYINT
IldIntegerType	SQL_C_LONG	CINTTYPE	SQLINT4	SQL_C_INTEGER
IldRealType	SQL_C_DOUBLE	CDOUBLETYPE	SQLFLT8	SQL_C_DOUBLE
IldStringType	SQL_C_CHAR	CCHARTYPE	SQLCHAR	SQL_C_CHAR
IldDateType IldDateTimeType	SQL_C_CHAR	CCHARTYPE	SQLDATETIME	SQL_C_CHAR
IldMoneyType	SQL_C_DOUBLE	CDOUBLETYPE	SQLMONEY	SQL_C_DOUBLE
IldLongTextType	SQL_C_CHAR	CLOCATOR	SQLTEXT	SQL_C_CHAR
IldBinaryType	SQL_C_BINARY	CLOCATOR	SQLBIT	SQL_C_BINARY
IldObjectType	-	CROWTYPE	-	-
IldCollectionType	-	CCOLLTYPE	-	-
IldCursorType	-	-	-	-
IldRefType	-	-	-	-
IldCLOBType	SQL_CLOB_LOCATOR	CLOCATOR	-	-
IldBLOBType	SQL_BLOB_LOCATOR	CLOCATOR	-	-

IBM ILOG DB Link (Continued)	Oracle	Sybase
IldByteType	SQLT_INT	CS_INT
IldIntegerType	SQLT_INT	CS_INT
IldRealType	SQLT_FLT	CS_FLOAT
IldStringType	SQLT_STR	CS_CHAR

IBM ILOG DB Link (Continued)	Oracle	Sybase
IldDateType IldDateTimeType	SQLT_STR	CS_CHAR
IldMoneyType	SQLT_FLT	CS_FLOAT
IldLongTextType	SQLT_STR	CS_CHAR
IldBinaryType	SQLT_LBI	CS_BINARY
IldObjectType	SQLT_NTY	-
IldCollectionType	SQLT_NTY	-
IldCursorType	SQLT_RSET	-
IldRefType	SQLT_REF	-
IldCLOBType	SQLT_CLOB	-
IldBLOBType	SQLT_BLOB	-

The following special features of the Input Mode are described separately:

- *Date As String*
- *Numeric As String*
- *MS SQL Server Limitation*

Date As String

When the *date as string* feature is turned off, the IBM ILOG DB Link `IldDateTime` type values are sent using different database client API type symbols.

DB2	SQL_C_TYPE_TIMESTAMP
Informix	CDTIMETYPE
ODBC	SQL_C_TIMESTAMP
Oracle	SQL_TIMESTAMP
Sybase	CS_DATETIME_TYPE

Numeric As String

A similar change happens when the *numeric as string* feature is turned on:

DB2	`SQL_C_NUMERIC`
Informix	`CCHARTYPE`
ODBC	`SQL_C_CHAR`
Oracle	`SQLT_STR`
Sybase	`CS_NUMERIC`

MS SQL Server Limitation

MS SQL Server does not automatically convert string values into integer values; the application must use the SQL function `convert`:

```
cout << "Data insertion : " << endl;
const char* insertStr =
  ((!strncmp(dbms->getName(), "oracle", 6)
    || !strcmp(dbms->getName(), "sqlbase")) ?
   "insert into ATABLE values(:1, :2)"
   : ((!strcmp(dbms->getName(), "mssql") ||
     !IldStrNICaseCmp(dbOdbc, "Microsoft SQL Server", 20)) ?
     "insert into ATABLE values (convert(numeric(28, 9), ?), ?)"
     : "insert into ATABLE values(?, ?)"));
```

C H A P T E R

2

Configuration Issues

This chapter is divided as follows:

- *Environment Variables* describes the environment variables required by IBM® ILOG® DB Link for each RDBMS.
- *Configuration File* explains how to load the configuration files that are necessary to your working environment.
- *Configuration Features* deals with configuration file issues and some basic configuration features.
- *Asynchronous Processing Mode* describes the principle of asynchronous processing in IBM ILOG DB Link, as well as the changes brought about by this mode.
- *Server Information* describes what implementation information items can be retrieved from the server.

Environment Variables

Depending on the operating system and the target RDBMS, some environment variables must be set. Some configurations also require that you add the appropriate paths to your environment. These considerations are described for the following RDBMSs:

- *DB2*
- *Informix*
- *MS SQL Server*
- *Oracle*

DB2

The environment variables `DB2DIR` and `DB2INSTANCE` must be set.

Informix

- UNIX® systems require the following variables:
  - `INFORMIXDIR`, which locates the Informix client installation.
  - `INFORMIXSERVER`, which provides the name of the default server.
  - `DELIMIDENT=y`, which must be defined if your application is to use delimited identifiers, because IBM ILOG DB Link libraries are compiled with this variable.
  - If you use shared libraries, add `$INFORMIXDIR/lib` and `$INFORMIXDIR/lib/esql` to the appropriate path variable (`PATH` or `LD_LIBRARY_PATH` or `SHLIB_PATH`).
- On PCs, the use of the Informix Utility `setnet32` sets all necessary variables in the registry.

MS SQL Server

All needed information is set in the registry upon installation.

Oracle

- On UNIX systems, the environment variable `ORACLE_HOME` must be set. If you use shared libraries, add `$ORACLE_HOME/lib32` if running in 32bits, or `$ORACLE_HOME/lib` if runnning 64bits , to the shared libraries path variable.
- On PCs, the variable `ORACLE_HOME` and numerous other values are set in the registry upon installation, and `%ORACLE_HOME%\bin` is added to the `PATH` variable.

Configuration File

If you use the dynamic load feature, IBM® ILOG® DB Link looks for a configuration file that describes the drivers allowed for the current platform before establishing a connection.

> ***Note:*** *When the driver is linked statically, this file is not used.*

The following items concerning this configuration file are described:

- *Format*
- *Location*
- *Resolving Library Names and Loading Libraries*

Format

The default configuration file `dblink.ini` is included in the standard distribution. This file contains a single section, `[dblink]`, which lists all available drivers with the following format:

```
[<dblink>]
<database name>=<library name>
```

where:

- `<database name>` is one of the database system names listed in section *Connection Arguments* on page 44,
- `<library name>` is the root of the IBM ILOG DB Link driver-library name. See *Resolving Library Names and Loading Libraries*.

Location

- On PCs, the configuration file is searched for first in the executable directory, and then in the Windows® root directory.

  A first scan looks for a file named after the executable name. For example, if the application name is `myApp`, IBM ILOG DB Link first looks for a configuration file named `myapp.ini`. If no such file is found, IBM ILOG DB Link looks for the default file `dblink.ini`.

- On UNIX, the configuration file is first searched using the contents of the environment variable `ILDHOME`, if it is defined. The suffix `.ini` is appended to the variable contents. If no such file is found, it is searched for locally. If it is still not found, IBM ILOG DB Link first looks for the default file `dblink.ini` in the directory defined by `ILDHOME`, then locally.

Whichever the platform, if the entry is not found in the application configuration file `myapp.ini`, it will be searched for in the default file `dblink.ini`.

If the configuration file is not found using this method, DBLink will use the following hard-coded values:

- db2 = dbdb2
- db29x = dbdb29x
- informix9 = dbinf9
- mssql = dbmssql
- odbc = dbodbc
- oledb = dboledb
- oracle9 = dbora9
- oracle10=dbora10
- oracle11=dbora11
- sybase = ctsyb

Resolving Library Names and Loading Libraries

This step exists only when loading a driver dynamically.

The library name is built from the value of the entry in the configuration file that corresponds to the first argument passed to the inline function `IldNewDbms`. Depending on the operating system, a prefix can be added (`lib` under UNIX) and an extension is appended (`.so` for Solaris, Linux, AIX and Tru64 UNIX®, `.dll` for PCs, `.sl` for HP-UX).

The library is then loaded using the system library functions dedicated to that purpose. These functions and their behavior vary from one operating system to another.

- On PCs, the function `LoadLibrary` automatically searches the library using the contents of the environment variable `PATH`.
- In compliance to POSIX standard, on UNIX, the library is always searched in the directories declared in the variable `LD_LIBRARY_PATH`.

Configuration Features

This section describes the way date-and-time values and numeric values are handled. It also discusses the array modes whereby IBM ILOG DB Link sends or fetches several rows at a time. The following items are described:

- *Date As String*
- *Numeric As String*
- *Numeric As Object*
- *Array Bind*
- *Array Fetch*

Date As String

Date-and-time related values can be sent and retrieved as strings. This entails a dependence on the RDBMS configuration parameters and `LOCALE` settings. The behavior with date-and-time column data types can be changed. The default behavior ensures compatibility with older versions, but the new behavior allows you to use objects to handle date-and-time values.

- To turn off the default behavior for all `IldRequest` objects created from a specific `IldDbms` object or to set a specific `IldRequest` object to handle date-and-time related values as objects, use the member function `IldIldBase::setStringDateUse`, with its argument set to `IlFalse`.

- To find the current setting, use the member function `IldIldBase::useStringDate`, which returns a Boolean value.

Both functions are inherited from the common base class `IldIldBase`.

The error `ILD_TYPE_MISMATCH` is raised when an application tries to send or retrieve a date-and-time value as an object when the *date as string* feature is turned on. Likewise, the same error is raised if an application tries to send or retrieve a date-and-time value as a string —instead of as an object—when the *date as string* feature is turned off.

Numeric As String

Numeric data values (and decimal values when applicable) can be sent and retrieved as `double` values. However, this causes a loss in precision. The behavior with respect to exact numeric column data types can be changed.

The default behavior ensures compatibility with older versions, but you can change it to preserve exact precision for very large numbers by handling these values as strings.

- To turn off the default behavior for all `IldRequest` objects created from a specific `IldDbms` object or to set a specific `IldRequest` object to handle numeric values as strings, use the member function `IldIldBase::setStringNumericUse` with its argument set to `IlTrue`. To handle numeric values as numeric objects, use the member function with its argument set to `IlFalse`, as shown below:

```
{
    // Data selection using numeric objects
    request->setStringNumericUse(IlFalse);
}
```

- To find the current setting, use the member function `IldIldBase::useStringNumeric`. It returns a Boolean value.

Both member functions are inherited from the common base class `IldIldBase`. No error is raised if an application retrieves a numeric value as `double` when the *numeric as string* feature is turned on, but the returned value is irrelevant.

It is possible to bind a database numeric type variable as an IBM ILOG DB Link string variable. The conversion is handled silently.

Numeric As Object

The *numeric as string* feature has a drawback: since the application depends upon the current `LOCALE`, the fractional-part and thousands separators may change from one session to another.

To avoid that external dependency, IBM ILOG DB Link allows you to send and retrieve numeric and decimal values under object form. The class `IlNumeric` is intended for that purpose.

- To turn off the default behavior for all `IldRequest` objects created from a specific `IldDbms` object or to set a specific `IldRequest` object to handle numeric values as objects, use the member function `IldIldBase::setNumericUse` with its argument set to `IlTrue`.
- To find the current setting, use the member function `IldIldBase::useNumeric`, which returns a Boolean value.

  Both member functions are inherited from the common base class `IldIldBase`.
- To retrieve the numeric value of a select-list column in object form, use the function `IldRequest::getColNumericValue`. The error `ILD_TYPE_MISMATCH` is raised when this function is used and the *numeric as object* feature has not been turned on. Conversely, it is also an error to try to retrieve the value in string form if the feature is turned on.
- A parameter value can be set using the function `IldRequest::setParamValue` and retrieved by means of the function `IldRequest::getParamNumericValue`.

Array Bind

Array bind means that IBM ILOG DB Link sends several rows of parameter values each time a prepared query is executed.

- To set the *array bind* mode, pass the number of rows you want to be sent at a time.

  This number can be set as a default value for all `IldRequest` objects requested from one `IldDbms` object but it can be changed for any particular instance of `IldRequest` whenever needed.
- To set the default value for all newly created `IldRequest` objects, use the member function `IldDbms::setDefaultParamArraySize`, passing it a positive integer value as its argument. The new value is set to all cursors requested after that setting, but it is not changed for cursors already held by the application. The current default array bind size can be retrieved using the function `IldDbms::getDefaultParamArraySize`.
- For each `IldRequest` object, you can change the array size using the function `IldRequest::setParamArraySize` with a positive integer as its argument. To be effective, this setting must take place before the call to `IldRequest::parse` or `IldRequest::execute`.
- To get the array size, use `IldRequest::getParamArraySize` and to reset it, use `IldRequest::removeParamArraySize`.

For RDBMS whose API does not support this feature, IBM ILOG DB Link emulates it. This is the case of Informix, which supports the array bind mode only for `insert` statements within a transaction.

Array Fetch

Array fetch means that IBM ILOG DB Link fetches several rows at a time from the current result set and buffer the returned values. This optimizes network traffic by reducing the number of messages exchanged between the client application and the database server.

- To set the default value for all `IldRequest` objects requested from an `IldDbms` object, use the function `IldDbms::setDefaultColArraySize` with a positive integer as its argument. The cursors already held by the application are not affected by the setting.
- For each `IldRequest` object, this setting can be changed using the member function `IldRequest::setColArraySize` with a positive integer as its argument, or reset using `IldRequest::removeColArraySize`. To be effective, the setting must take place before the first call to `fetch`.
- To get the current array size, use `IldRequest::getColArraySize`.

With ODBC, the *array fetch* feature is available only if the driver has level 2 compliance.

For RDBMS whose API does not support this feature, IBM ILOG DB Link emulates it, but with no optimization effect on the network load.

Asynchronous Processing Mode

This section describes what this processing mode does, how it changes the application behavior, what drivers currently support this mode and what member functions use it. The following items are described:

- *Principle*
- *Important Behavior Change*
- *Drivers that Support Asynchronous Processing*
- *Functions that Use Asynchronous Processing*

Principle

When this mode is turned on, the execution of a query immediately returns control to the application.

The application must then be designed so as to call the function again with the very same arguments until the query completes. To test the completion status, the application calls the function `IldRequest::isCompleted`, which returns:

- `IlTrue` if:
  - an error was raised, or
  - the caller is inactive, or
  - the query is completed.
- `IlFalse` if the execution of the query is still in progress.

Important Behavior Change

When the asynchronous processing mode is turned on, only ONE query can be active at a time for a given connection. In this case, it is impossible to use two different `IldRequest` objects pertaining to the same connection (an `IldDbms` object) simultaneously. However, it is possible to allocate several `IldRequest` objects for the same connection, and use any of these requests as soon as the previous operation is completed.

Drivers that Support Asynchronous Processing

The function `IldDbms::isAsyncSupported` returns `IlTrue` when the driver supports the asynchronous processing mode. Currently these drivers are:

- `oracle9`, 10 and 11
- `sybase`
- `mssql`
- `odbc`, when the underlying ODBC driver supports it.

Functions that Use Asynchronous Processing

When the asynchronous processing mode is turned on, the following member functions must be checked for completion before accessing their results:

- `IldDbms` class
  - `IldDbms::readRelation`
  - `IldDbms::readRelationNames` (overloaded)
  - `IldDbms::readRelationOwners`
  - `IldDbms::subscribeEvent`
  - `IldDbms::unSubscribeEvent`
- `IldRelation` class
  - `IldRelation::getForeignKeys`
  - `IldRelation::getIndexes`
  - `IldRelation::getPrimaryKey`
  - `IldRelation::getSpecialColumns`

    For these four functions, check with:

```
rel->getDbms().isCompleted()
```

For these two classes, the returned values are significant if, and only if, the following test holds:

```
dbms->isCompleted() && !dbms->isErrorRaised()
```

- `IldRequest` class
  - `IldRequest::execute` (overloaded)
  - `IldRequest::fetch`
  - `IldRequest::insertBinary`
  - `IldRequest::insertLongText`
  - `IldRequest::getLargeObject`
  - `IldRequest::getLargeObjectChunk`
  - `IldRequest::parse`
  - `IldRequest::startGetLargeObject`

Server Information

Information about implementation is retrieved by calling the function `IldDbms::getInfo`. The first argument of this function takes its value in the enumeration type `IldInfoItem`.

The enumeration is defined in the file `ildconst.h`. It uses the CLI-defined symbols, prefixed with the IBM ILOG DB Link prefix `Ild`, and complies with the CLI numeric values of these symbols. The values returned by the function `getInfo` also are compliant with the CLI specifications except for `IldOuterJoinCapabilities`, where a number is returned instead of a one-character string.

For information items whose value is a string, the server usually returns the exact value to the appropriate query, but for those with numeric values, these values are translated to symbols derived from the CLI standard.

The following table lists the type of the output argument from the function `IldDbms::getInfo` that is used to return the value of the item.

Table 2.1 *Server Information Items*

Symbol	CLI Code	Item Value	Possible Values
`IldAlterTable`	86	integer	Depending on the RDBMS SQL implementation, this item returns a value resulting from the logical OR combination of: ◆ `IldAlterTableAddColumn` (1), ◆ `IldAlterTableDropColumn` (2), ◆ `IldAlterTableAlterColumn` (4), ◆ `IldAlterTableAddConstraint` (8), and ◆ `IldAlterTableDropConstraint` (16).
`IldCatalogName`	10003	string	
`IldCollatingSequence`	10004	string	
`IldCursorCommitBehavior`	23	integer	This item returns one of the following values, depending on the connected RDBMS: ◆ `IldCurBehaviorDelete` (0), ◆ `IldCurBehaviorClose` (1), or ◆ `IldCurBehaviorPreserve` (2).

Table 2.1 *Server Information Items (Continued)*

Symbol	CLI Code	Item Value	Possible Values
IldCursorSensitivity	10001	integer	This item returns one of the following values: ◆ IldCursorASensitive (0), ◆ IldCursorInSensitive (1), or ◆ IldCursorSensitive (2)
IldDataSourceName	2	string	
IldDataSourceReadOnly	25	string	
IldDBMSName	17	string	
IldDBMSVersion	18	string	
IldDefTransactionIsolation	26	integer	This item returns one of the following values: ◆ IldTransIsolReadUncommitted (1), ◆ IldTransIsolReadCommitted (2), ◆ IldTransIsolRepeatableRead (3), or ◆ IldTransIsolSerializable (4)
IldDescribeParameter	10002	string	
IldFetchDirection	8	integer	This item returns a logical OR operation between all supported fetch directions. For databases that do not support scrollable cursors, the only possible value will be IldFetchDirectionNext (1). ◆ IldFetchDirectionNext (1), ◆ IldFetchDirectionFirst (2), ◆ IldFetchDirectionLast (4), ◆ IldFetchDirectionPrior (8), ◆ IldFetchDirectionAbsolute(16), and ◆ IldFetchDirectionRelative (32)
IldGetDataExtension	81	integer	For this item, the returned value is always the sum of: ◆ IldGetDataAnyColumn (1) and ◆ IldGetDataAnyOrder (2) because IBM ILOG DB Link implements these capabilities for all supported RDBMSs.

Table 2.1 *Server Information Items (Continued)*

Symbol	CLI Code	Item Value	Possible Values
IldIdentifierCase	28	integer	This item returns one of the following values: ◆ IldIdentifierUpper (1), ◆ IldIdentifierLower (2), ◆ IldIdentifierSensitive (3), or ◆ IldIdentifierMixed (4)
IldIntegrity	73	string	
IldMaxCatalogNameLength	34	integer	
IldMaxColumnsInGroupBy	97	integer	
IldMaxColumnsInOrderBy	99	integer	
IldMaxColumnsInSelect	100	integer	
IldMaxColumnsInTable	101	integer	
IldMaxColumnNameLength	30	integer	
IldMaxConcurrentActivities	1	integer	
IldMaxCursorNameLength	31	integer	
IldMaxDriverConnections	0	integer	
IldMaxIdentifierLength	10005	integer	
IldMaxSchemaNameLength	32	integer	
IldMaxStatementLength	105	integer	
IldMaxTableNameLength	35	integer	
IldMaxTablesInSelect	106	integer	
IldMaxUserNameLength	107	integer	
IldNullCollation	85	integer	This item returns one of the following values, depending on whether the RDBMS collates null values first or last in the result sets: ◆ IldNullCollateHigh (0) or ◆ IldNullCollateLow (1)
IldOrderByColumnsInSelect	90	string	

Table 2.1 *Server Information Items (Continued)*

Symbol	CLI Code	Item Value	Possible Values
`IldOuterJoinCapabilities`	115	integer	This item returns a logical OR operation between all supported outer join capabilities: ◆ `IldOuterJoinLeft` (1), ◆ `IldOuterJoinRight` (2), ◆ `IldOuterJoinFull` (4), ◆ `IldOuterJoinNested` (8), ◆ `IldOuterJoinNotOrdered` (16), ◆ `IldOuterJoinInner` (32), or ◆ `IldOuterJoinAllOps` (64)
`IldScrollConcurrency`	43	integer	This item returns a logical OR operation between all supported options: ◆ `IldScrollReadOnly` (1), ◆ `IldScrollLock` (2), ◆ `IldScrollOptRowver` (4), and ◆ `IldScrollOptValues` (8)
`IldServerName`	13	string	
`IldSpecialCharacters`	94	string	
`IldTransactionCapable`	46	integer	This item returns one of the following values: ◆ `IldTransCapableNone` (0), ◆ `IldTransCapableDML` (1), ◆ `IldTransCapableAll` (2), ◆ `IldTransCapableDDLCommit` (3), or ◆ `IldTransCapableDDLIgnore` (4).
`IldTransactionIsolationOpt`	72	integer	This item returns one of the following values: ◆ `IldTransIsolReadUncommitted` (1), ◆ `IldTransIsolReadCommitted` (2), ◆ `IldTransIsolRepeatableRead` (3), or ◆ `IldTransIsolSerializable` (4)
`IldUserName`	47	string	

For all items that return an integer value, the return value is `0` if the value is unknown. When an error is raised during the retrieval of an information item, the integer argument is set to `-1` and the string argument is set to the empty string (the first character is a `null` character).

C H A P T E R

3

Sessions & Connections

An application can be designed to communicate with several RDBMSs at a time, each communication being represented by a connection that makes up the active part of a session. A connection can be closed and reopened using a different authentication.

IBM® ILOG® DB Link implements the session concept through the class `IldDbms`, which is also the repository for the connection control.

`IldDbmsModel`, a twin class to `IldDbms`, is provided to develop drivers for RDBMS that aren't currently supported by IBM ILOG DB Link. It supports the same functionalities as `IldDbms`, plus the ability to be derived. So, in this manual, we describe the `IldDbms` class, which is the main one. The few differences between the two twin classes are itemized in a subsection.

This chapter is divided as follows, to reflect what this class allows:

- *Connection Handling through IldDbms Objects*—Connecting, disconnecting, reconnecting.
- *Accessing the Database Schema* for database schema descriptions—Several classes of descriptors exist to describe the various entities contained in the target database.
- *Data Definition Language (DDL)*—Executing DDL and DML statements.
- *Transaction Control*—All transactional operations, such as initiating, committing, or rolling back a transaction.
- *Cursor Allocation*—Cursors are created on demand and then cached in a pool for reuse.

- *Extending the IldDbms Class*—This class cannot be derived, but you can extend its functionalities under certain conditions. If you need to derive the `IldDbms` class, you must use its twin class, `IldDbmsModel`.
- *Use Notification*—This feature lets you receive notifications from the RDBMS asynchronously.
- *Differences between IldDbms and IldDbmsModel Classes*

Connection Handling through IldDbms Objects

With IBM ILOG DB Link, an application connects to an RDBMS through an object of the class `IldDbms`. In fact, the first thing that IBM ILOG DB Link must do before an interactive session can be initiated with a database server is to create such an `IldDbms` object.

This section explains how to create, manipulate, and delete `IldDbms` objects. It is divided as follows:

- *Initiating a Session or a Connection*
- *Creating IldDbms Objects*
- *Session Configuration*
- *Disconnecting and Reconnecting*
- *Number of Connections*
- *Destroying IldDbms Objects*

Initiating a Session or a Connection

The only way to initiate a session or a connection is to create an `IldDbms` object. To do this, use the inline function `IldNewDbms`. (This entry point is defined as an inline global function in the header file `dblink.h`.) As a consequence, a connection is activated. It can be closed by calling the function `IldDbms::disconnect` and reopened later by calling the function `IldDbms::connect`.

> ***Note:*** *Several connections can be active at the same time. Their number is limited only by the server configuration (**not** by IBM ILOG DB Link itself).*

When the `IldDbms` object is deleted, the connection is automatically closed and all dependent objects are deleted.

If the driver fails to establish the initial connection, an object from the class `IldErrorDbms` is returned.

Warning: *The class* `IldErrorDbms` *is not documented.*

There are five cases where an error of type `IldErrorDbms` can be returned and only one where an instance of `IldErrorRequest` is returned. All cases where an `IldErrorDbms` error is returned are handled by the driver manager in the function `IldAllocConnect`. Only the member function `IldDbms::getFreeRequest` may return an `IldErrorRequest` error, when the connection is not established and the error handler fails to fix it.

Error-Raising Conditions

- **Null or empty strings as arguments**

  The arguments passed to the function `IldNewDbms` must not be null or empty strings. Otherwise, an `IldErrorDbms` error is returned.

  See *Connection Arguments* for details about these two arguments.

- **Driver not linked or not found**

  - When the drivers are linked *statically*, the driver whose name is passed as first argument to the function `IldNewDbms` must be found. Otherwise, an instance of `IldErrorDbms` is created, initialized in an error state—that is, the function `IldIldBase::isErrorRaised` returns `IlTrue`—and returned. (This function is inherited from the common base class `IldIldBase`.)
  - When the drivers are loaded *dynamically* and an entry with the proper name is missing in the `dblink.ini` file, the same behavior occurs.

- **Memory allocation failure**

  When the driver entry point is called and fails to return a valid address, the driver manager creates and returns an `IldErrorDbms` error.

- **Unknown driver**

  When using the dynamic load feature, if the RDBMS name is not found in the `[dblink]` section of the configuration file, the driver manager returns an `IldErrorDbms` error.

- **Improper driver found**

  When using the dynamic load feature, if the driver library is loaded properly but the entry point cannot be found, the driver manager allocates and returns an `IldErrorDbms` error.

- **Unconnected**

  When the connection is not established or has been closed, and the error handler did not attempt to re-establish it or this attempt failed, the `IldDbms::getFreeRequest` function allocates and returns an `IldErrorRequest` object.

The only time that the function `IldNewDbms` can return a null value is when no object can be allocated because the application ran out of memory.

Creating IldDbms Objects

There is no public constructor for objects of the class `IldDbms`. To create `IldDbms` objects, use the inline function `IldNewDbms`. This function is defined in your application as soon as the file `dblink.h` is included. Its code is modified by the compile-time flags you define.

If your application is linked in dynamic load mode, no RDBMS-specific compile-time flag is needed.

Throughout this manual, multiple examples show you how to use the function `IldNewDbms`.

```
{
    cout << "Connecting to: " << IldDbmsName << endl;
    IldDbms* dbms = IldNewDbms(IldDbmsName, IldConString);
    if (!dbms) cout << "Out  of memory" << endl;
    if (dbms->isErrorRaised()) {
        IldDisplayError("Creating Dbms: ", dbms);
        delete [] queryBuffer;
        delete dbms;
        return 1;
    }
}
```

> ***Warning:*** *Even though the allocation may be successful, the creation of the IldDbms object may still fail. To make sure the connection is successful, you must* ***a)*** *Check that* `IldNewDbms` *returns a non-null pointer;* ***b)*** *Use the member function* `IldIldBase::isErrorRaised` *to check whether the* `IldDbms` *object was successfully allocated.*

Automatic Connection

Creating an `IldDbms` object connects you to the RDBMS using the values passed to the function `IldNewDbms`. This initial connection is required to test whether the server can be reached and is ready for communication.

Connection Arguments

The function `IldNewDbms` takes two arguments, which are the DB Link name of the database system and the connection string.

- The first argument is the *name of the database system* as known by IBM ILOG DB Link. It must have one of the following values:
  - `db2`
  - `db29x`
  - `informix9`
  - `mssql`
  - `odbc`
  - `oledb`
  - `oracle9`
  - `oracle10`
  - `oracle11`
  - `sybase`

  These names are all lowercase and must be entered exactly as shown. Any other names are illegal when using IBM ILOG DB Link-supported drivers.

  If the name you pass does not match one from the above list, IBM ILOG DB Link raises either the error `ILD_UNKNOWN_RDBMS`, indicating that it does not recognize the RDBMS, or `ILD_LIB_NLNKD` if the application did not link the driver statically.
- The second argument is the *connection string*. It must comply with a format that depends on the target RDBMS.

Connection String Format

The format and contents of a connection string depend on the target RDBMS:

- DB2: `[<user>]/[<password>]/<database name>`
- Informix: `[<user>]/[<password>]/<database name>[@<server>]`
- MS SQL Server: `<user>/<password>/<database name>/<server name>`
- ODBC: `<data source name>/[<user>]/[<password>]`
- Oracle: `[<user>]/[<password>][@<service>]`
- Sybase: `<user>/<password>/<database name>/<server name>`

Values enclosed in square brackets are optional.

> ***Note:*** *With ODBC, you cannot pass the database name in the connection string. It can be set through the* `odbc.ini` *file. Also, the slash marks ('/') are mandatory.*
>
> *The ODBC driver also supports a connection string following the format:*
>
> *"*`DRIVER= ...; DBQ=...`*"*

If the second argument passed to the function `IldNewDbms` does not comply with the appropriate format, IBM ILOG DB Link raises the error `ILD_BAD_DB_SPEC` indicating that the connection string is not valid for this RBDMS. Nothing can be done using that `IldDbms` object until a valid connection is established.

Session Configuration

Default Error Reporter

When you create an `IldDbms` object, it is associated with a new `IldErrorReporter` object. The default reporter is not accessible to the application. As a consequence, the function `IldDbms::getErrorReporter` returns a null pointer if no user-derived error reporter has been set.

To customize error handling, you can create your own error reporter class and instantiate it for the `IldDbms` object reporter using the function `IldDbms::setErrorReporter`.

Default Configuration

The default settings for a session configuration are the following:

- The *date as string* feature is turned on.
- The *numeric as string* feature is turned off.
- The *numeric as object* feature is turned off.
- The array size for the *array bind* and *array fetch* modes is set to 1.

Checking the Default Configuration of a Connection

Once you have created the first object of the class `IldDbms`, you can access the following configuration settings:

- To get the versions of the currently accessed RDBMS against which IBM ILOG DB Link was tested, use the function `IldDbms::getDbmsVersions`.
- To get the main version number of the supported RDBMS, use the member function `IldDbms::getDbmsVersion`.
- To get the information items obtained from the server, use `IldDbms::getInfo`.

None of these values can be changed, so they remain valid for all `IldDbms` objects your application creates.

Checking the Current Configuration of a Connection

In the current `IldDbms` object, several session-wide parameters are set. Table 3.1 shows what `IldDbms` member function you can use to check their values.

Table 3.1 *Checking the Current Configuration of a Connection*

Use this member function...	...to get this setting.	Default Value
`IldDbms::getName`	IBM ILOG DB Link name of the currently accessed RDBMS	
`IldDbms::getUser`	Name of the user who established the connection	
`IldDbms::getDatabase`	Name of the database used to establish the connection	
`IldDbms::getDefaultColArraySize`	Default size of the array used to fetch rows when the SQL statement executed is a `select` query	`1`[1]
`IldDbms::getDefaultParamArraySize`	Default size of the array used to send rows of parameter values	`1`[2]
`IldIldBase::useStringDate`	*Date as string* feature	`IlTrue`
`IldIldBase::useStringNumeric`	*Numeric as string* feature	`IlFalse`
`IldIldBase::useNumeric`	*Numeric as object* feature	`IlFalse`

[1] This default value means that rows will be fetched one by one. It is automatically used at creation time for all objects of the class `IldRequest` that depend on that `IldDbms` object.

[2] This default value means that the parameter rows will be sent one by one. Like the fetch array size, this value is set at creation time for all objects of the class `IldRequest` that depend on that `IldDbms` object.

Changing the Configuration of a Connection

You can change the settings in your current connection configuration, as shown in Table 3.2:

Table 3.2 *Changing the Settings of the Current Connection Configuration*

Use	To
`IldDbms::setDefaultColArraySize`	Change the default fetch array size[1]
`IldDbms::setDefaultParamArraySize`	Change the default parameter array size
`IldIldBase::setStringDateUse`	Turn off the *date as string* feature

Table 3.2 *Changing the Settings of the Current Connection Configuration*

Use	To
`IldIldBase::setStringNumericUse`	Turn on the *numeric as string* feature
`IldIldBase::setNumericUse`	Turn on the *numeric as object* feature

[1] The new values are inherited at creation time by all `IldRequest` objects built *after* one of these functions has been called. Array size values for objects that were created before the call remain unchanged.

Disconnecting and Reconnecting

To disconnect from an RDBMS, you must call the function `IldDbms::disconnect` as follows:

```
{
cout << "Disconnecting from: " << argv[1] << endl;
if (!dbms->disconnect())
    IldDisplayError("Disconnection failed: ", dbms);
}
```

> ***Note:*** *The error reporter of the* `IldDbms` *object is inherited by all subsequently created* `IldRequest` *objects.*

This function:

- deletes all its attached `IldRequest` objects;
- deletes all its attached schema entity description objects;
- closes the connection to the RDBMS.

Once you have disconnected, you cannot create an `IldRequest` object from that same `IldDbms` object. Any such attempt raises the error `ILD_DBMS_NOT_CONNECTED` and an `IldErrorRequest` object is returned to the calling application.

With a disconnected `IldDbms` object, you can reconnect to any database from the same database system by calling the member function `IldDbms::connect`. Its argument is a connection string of the same format as the second argument passed to the function `IldNewDbms`. If the `IldDbms` object was not properly disconnected prior to your call to `connect`, the error `ILD_ALREADY_CONNECTED` is raised, indicating that the current object is still in use or has not been properly disconnected yet.

```
{
cout << "New connection to: " << argv[1] << endl;
if (!dbms->connect(argv[2]))
    IldDisplayError("Reconnection failed: ", dbms);
}
```

The member function `IldDbms::disconnect` is used in the example `testerr` where an attempt to connect twice is made deliberately. The user-defined error reporter forces a disconnection to enable the new connection to be made:

```
{
case ILD_ALREADY_CONNECTED:
   cout << endl
   << "USER WARNING: already connected to: "
   << dbms->getDatabase()
   << endl;
   dbms->disconnect();
   // The connection will be performed by DB Link itself.
   break;
}
```

Number of Connections

IBM ILOG DB Link has no built-in limitation to the number of connections an application can create. The RDBMS itself raises an error when its maximum number of connections is reached. This maximum may be configured.

The member function `IldDbms::getNumberOfActiveConnections` returns the number of `IldDbms` objects created.

Destroying IldDbms Objects

All `IldDbms` objects created by an application must be destroyed before the application exits to avoid memory leaks and possible dangling connections to the RDBMS.

The `IldDbms` destructor has the same effect on attached `IldRequest` objects and schema entity description objects as a call to the function `IldDbms::disconnect`. All these objects are destroyed, hence, there is no need to delete them. IBM ILOG DB Link takes care of deleting them before deleting the `IldDbms` object itself.

> ***Important:*** *Objects of classes derived from* `IldDbmsModel` *DO NOT behave in this way:* `IldRequestModel` *derived objects MUST be explicitly separately deleted.*

Accessing the Database Schema

IBM ILOG DB Link offers several functions to access the database schema or catalog.

A schema entity is any autonomous structure in a schema: this includes tables, views, stored procedures, user-defined data types, and synonyms. However, indexes or primary keys are not schema entities because they cannot be described independently of a table.

Most entities belonging to a database schema can be described by means of special descriptors. All schema entity descriptors are instances of classes derived from the class `IldSchemaEntity`:

- `IldRelation` for tables and views
- `IldCallable` for procedures and functions
- `IldSynonym` for synonyms
- `IldADTDescriptor` for user-defined data types (supported only when connected to Object-Relational Data Base Management Systems).

It is also possible to get only the entity names and owner names. Names are returned as arrays of character strings by the following member functions of the class `IldDbms`:

- `IldDbms::readRelationNames`
- `IldDbms::readProcedureNames`
- `IldDbms::readSynonymNames`
- `IldDbms::readAbstractTypeNames`

All these functions take the owner name as an optional argument to restrict the returned names to the entities that belong to that owner. These items are described in the following order:

- *Schema Entity Types*
- *Schema Entity Names and Owners*
- *Tables and Views*
- *Procedures and Functions*
- *Synonyms*
- *Abstract Data Types*
- *Table Privileges*

Schema Entity Types

Any schema entity for which a descriptor class exists has an identifier in the enumeration type `IldEntityType` declared in the file `ild.h`.

For any descriptor derived from `IldSchemaEntity`, the actual type of the descriptor can be retrieved by calling the function `IldSchemaEntity::getEntityType`.

These descriptors have only one possible identifier, except for the table or view descriptor `IldRelation`, which can be identified by `IldTableEntity` or `IldViewEntity`, as summarized in Table 3.3.

Table 3.3 *Schema Entity Descriptors*

Descriptors	Identifiers
Table	`IldTableEntity`
View	`IldViewEntity`
Procedure or function	`IldCallableEntity`
Synonym	`IldSynonymEntity`
User-defined type (abstract data type)	`IldADTEntity`
Error	`IldUnknownEntity`

Schema Entity Names and Owners

- The names of all owners of any entity in the schema can be retrieved by calling the function `IldDbms::readOwners`.
- The names and owners of any entity type in the schema can be retrieved as a cursor using the function `IldDbms::readEntityNames`. This function takes the entity type as its first argument and, optionally, an owner's name as its second argument, and returns a result set in form of a fetch-ready `IldRequest` object.

Names and Owners of Tables or Views

If you are interested only in table names, you can use the member function `IldDbms::readRelationNames`. It returns an array of all table and view names found in the schema. It is your responsibility to delete the array returned and the strings it contains, preferably using the function `IldDbms::freeNames`:

```
{
   char** names = dbms->readRelationNames();
   if (names) {
      cout << "All relation names:" << endl;
      for (int i = 0; names[i] != 0 ; i++)
         cout << "   " << names[i] << endl;
      dbms->freeNames(names);
   }
}
```

Some database management systems, such as Oracle, allow different users to create different tables with the same name. In this case, the array returned contains the same name several times. The second argument is optional and changes the behavior of the function:

- If the `user` argument is specified, only the names of the tables that belong to the given user name are returned.
- If not, all table names from the current schema are returned.

```
{
   cout << endl << "Give a USER name[CR for no USER]: ";
   cin.getline(str, 100);
   if (str[0]) {
   // We got a user.
   names = dbms->readRelationNames(str);
   if (names) {
      cout << "Relation names belonging to " << str << ": "
         << endl;
      for (int i = 0 ; names[i] != 0 ; i++) {
         cout << "   " << names[i] << endl;
      }
      dbms->freeNames(names);
   }
   else
      cout << "  NONE " << endl;
   }
}
```

If you want to know all table names and their owner names, the overloaded member function `IldDbms::readRelationNames` returns an array of the table names and sets its parameter to an array of the owner names. It is your responsibility to delete both arrays and the strings they contain.

Procedure Names

All procedure and function names are returned by the function `IldDbms::readProcedureNames`. If the optional `user` argument is specified, this function returns only the names of the procedures and functions that belong to that user.

Synonym Names

All synonym names are returned by the function `IldDbms::readSynonymNames`. If the optional `user` argument is specified, this function returns only the names of the synonyms that belong to that user.

> ***Note:*** *This function is not supported for DB2, MS SQL Server, ODBC, and Sybase, which do not have the notion of synonyms.*

Abstract Data Type Names

All abstract data type names are returned by the function `IldDbms::readAbstractTypeNames`. If the optional `user` argument is specified, this function returns only the names of the abstract data types that belong to that user.

> ***Note:*** *This function is only supported for ORDBMS (Object-Relational Data Base Management Systems).*

Tables and Views

Within IBM ILOG DB Link, a database table or view is described as an object of the class `IldRelation`. Such objects can be created by calling one of the following functions:

- `IldDbms::readRelation`: This function takes the table or view name as its first argument, and the owner name as its optional second argument.

  This function does not cache the returned object, which, therefore, must be deleted by the application.

  An overloaded version of this function takes only one argument, namely the numerical identifier of the table or view. This second version is not supported by all RDBMS.

> ***Warning:*** *When a table description is returned by this function, it is not possible to get its keys and indexes. If the application needs to hold a table description that is not attached to a connection, it should get it from the next function, query the keys and indexes, and then ask the description to be detached from the connection using the function* `IldDbms::removeRelation`.

- `IldDbms::getRelation`. This function takes the table name as its first argument and an owner name as its optional second argument. This function adds the created object to the cache managed by the caller. Therefore, the object can be accessed later without querying the server and can be deleted automatically when the caller is destroyed.

  An overloaded version of this function takes only one argument, namely the numerical identifier of the table or view. This second version is not supported by all RDBMSs.

> ***Warning:*** *Some database systems, such as Oracle, allow different users to own different tables with the same name. With such systems, it is important to supply the* `user` *parameter. Otherwise, IBM ILOG DB Link builds the* `IldRelation` *object based on the first row returned from the database server and ignores the others.*

```
{
cout << "Trying to retrieve an unknown relation: " << endl;
IldRelation* relation = dbms->getRelation("ATABLE");
if (!relation)
   relation = dbms->getRelation("atable", "");
if (!relation)
   // We print the error message.
   if (dbms->isErrorRaised())
      IldDisplayError(intentErr, dbms);
}
```

Types

IBM ILOG DB Link returns different types of relations, depending on the RDBMS you are connected to. The available types are `IldTableEntity` and `IldViewEntity`.

The meaning of these symbols is straightforward. Depending on the target RDBMS, loading the schema may create 0 or n `IldRelation` objects of type `IldView` Entity

Table Characteristics

Objects of the class `IldRelation` can be accessed to get the table characteristics, as shown in Table 3.4:

Table 3.4 *Table Descriptors*

Use this function...	To get the following descriptor
`IldSchemaEntity::getEntityType`	Type of the table
`IldRelation::getCount`	Number of columns
`IldSchemaEntity::getName`	Table name
`IldSchemaEntity::getOwner`	Table owner
`IldSchemaEntity::getDbms`	Related `IldDbms` object
`IldRelation::getPrimaryKey`	Primary key, if any
`IldRelation::getForeignKeys`	Foreign keys, if any
`IldRelation::getIndexes`	Indexes, if any
`IldRelation::getSpecialColumns`	Special columns (—that is, the columns that uniquely identify one row in the table)

The global function `IldPrintRelation` in the `ildutil.cpp` sample file shows how to use these member functions. See *Relation Searching* on page 131 for details.

With the exception of the `IldDbms` object, no value returned by these functions can be modified.

Columns

From an `IldRelation` object, you can reach its column descriptions. IBM ILOG DB Link preserves the column order, except with ODBC, where the column order is not specified.

A column description includes its name, size, IBM ILOG DB Link type, and native type name, as well as the flag indicating whether it accepts null values. Use the following member functions to get these attributes:

- **Name**: `IldRelation::getColName(IlUShort)`.
- **Size**: `IldRelation::getColSize(IlUShort)`. The column size is always in bytes. It is the actual size (the maximum size for `CHAR` and `VARCHAR` database types) used by IBM ILOG DB Link to store or send the data values.

> ***Note:*** *The LOB-type columns do not follow this rule. Thus, the value returned by the function* `getColSize` *is not meaningful for such columns.*

- **DB Link type**: `IldRelation::getColType(IlUShort)`. This function returns a value from the enumeration `IldColumnType`.
- **Native SQL type name**: `IldRelation::getColSQLType(IlUShort)`. This function returns the name of the native SQL data type on the server.
- **Null-values flag**: `IldRelation::isColNullable(IlUShort)`

```
{
   for (i = 0; i < nbColumns; i++) {
      ostrstream ostr(ColumnSizeStr, 32);
      ostr << relation.getColSize(i) << ends;
      ItemsArray[0]._buffer = (char*)relation.getColName(i);
      ItemsArray[1]._buffer = (char*)relation.getColSQLType(i);
      ItemsArray[3]._buffer = (relation.isColNullable(i)
                               ? "true" : "false");
      cout << IldFormatLine(4, ItemsArray, IlFalse) << endl;
   }
}
```

All these member functions take a column number as the argument. A valid column number is:

- greater than or equal to 0, and
- strictly less than the value returned by `IldRelation::getCount`.

Keys, Indexes, and Special Columns

The primary key, foreign keys, indexes, and special column descriptors are not created at the same time as the table description. They are retrieved from the database only the first time they are accessed, using the `IldRelation` functions `IldRelation::getPrimaryKey`, `IldRelation::getForeignKeys`, `IldRelation::getIndexes`, and `IldRelation::getSpecialColumns`.

If no keys of that type exist, the server is not queried again on the next call to one of these functions.

Procedures and Functions

Within IBM ILOG DB Link, a database procedure or function is described as an object of the class `IldCallable`. Such objects can be created by calling one of the following member functions:

- `IldDbms::readProcedure`: This function takes the procedure or function name as its first argument and an owner name as the optional second argument. The returned object is not cached. Therefore, it is the application's responsibility to delete it.

  An overloaded version of this function takes only one argument, namely the numerical identifier of the procedure or function. This second version is not supported by all RDBMSs.

- `IldDbms::getProcedure`: This function takes the procedure or function name as its first argument and an owner name as the optional second argument. The created object is added to the cache managed by the caller. Therefore, the object can be accessed later without querying the server and can be deleted automatically when the caller is destroyed.

  An overloaded version of this function takes only one argument, namely, the numerical identifier of the procedure or function. This second version is not supported by all RDBMSs.

SQL Type of the Object

The `IldCallable` object returned can represent either a stored procedure or a stored function. To differentiate between the two, a call to `IldCallable::isProcedure` returns `IlTrue` if it is a procedure description, or `IlFalse` otherwise.

Arguments

The formal arguments to the procedure or function are represented by objects of the class `IldArgument`, which is derived from the class `IldDescriptor`. The number of arguments is returned by a call to `IldCallable::getArgumentsCount`.

An `IldArgument` object describes the argument by:

- **Its input or output mode**: One of the three functions `IldArgument::isInArgument`, `IldArgument::isOutArgument`, or `IldArgument::isInOutArgument` returns `IlFalse` while the other two return `IlTrue`.
- **A default value**: The member function `IldArgument::hasDefault` tells whether the argument has a default value or not.

Return Values

The return values of a function are described by instances of the class `IldDescriptor`. The number of returned values is given by `IldCallable::getResultsCount`.

Synonyms

Within IBM ILOG DB Link, a synonym is described as an object of the class `IldSynonym`. Such objects can be created by calling one of the following functions:

- `IldDbms::readSynonym`: This function takes the synonym name as its first argument and an owner name as the optional second argument. The returned object is not cached. Therefore, it is the application's responsibility to delete it.

  An overloaded version of this function takes only one argument, namely, the numerical identifier of the synonym. This second version is not supported by all RDBMSs.

- `IldDbms::getSynonym`: This function takes the synonym name as its first argument and an owner name as the optional second argument. The created object is added to the cache managed by the caller. Therefore, the object can be accessed later without querying the server and can be deleted automatically when the caller is destroyed.

  An overloaded version of this function takes only one argument, namely, the numerical identifier of the synonym. This second version is not supported by all RDBMSs.

> ***Note:*** *None of these functions are supported with DB2, MS SQL Server, ODBC, and Sybase because this concept does not exist in these database systems.*

Abstract Data Types

Within IBM ILOG DB Link, an abstract data type is described as an object of the class `IldADTDescriptor`. Such objects can be created by calling one of the following functions:

- `IldDbms::readAbstractType`: This function takes the abstract data type name as its first argument and an owner name as the optional second argument. The returned object is not cached. Therefore, it is the application's responsibility to delete it.

  An overloaded version of this function takes only one argument, namely, the numerical identifier of the abstract data type.

- `IldDbms::getAbstractType`: This function takes the abstract data type name as its first argument and an owner name as the optional second argument. The created object is added to the cache managed by the caller. Therefore, the object can be accessed later without querying the server and can be deleted automatically when the caller is destroyed.

An overloaded version of this function takes only one argument, namely, the numerical identifier of the abstract data type.

Note: *Both functions are only supported for ORDBMSs.*

Table Privileges

The privileges given to a specific table / view can be accessed by calling the function `IldDbms::readTablePrivileges`.

```
IldDbms::readTablePrivileges(const char* catalog,
                             const char* schema,
                             const char* table) ;
```

The first argument is used only by DBMSs that support three-part naming for tables (qualifier.owner.name). In particular, this is not supported by Oracle, which uses only schema.name.

Make sure that the table, schema, and catalog parameters are spelled with the correct case. The RDBMS case method must be used.

If no specific trustee is given to a table, the result set for this table will be empty.

For Sybase:

- table name is required,
- no wildcard-characters are allowed.

Data Definition Language (DDL)

When a DDL (Data Definition Language) statement must be executed, you can use the member function `IldDbms::execute`. This function behaves like the member function `IldRequest::execute`. If required, IBM ILOG DB Link silently allocates an `IldRequest` object and uses it.

This function can also be used for DML (Data Manipulation Language) statements, except for the `select` statement. If you need to know the number of processed rows —modified, inserted, or deleted— you must pass a valid pointer to an `IlInt` variable as the second argument to the function `IldDbms::execute` (the first argument being the text of the SQL statement). This is the only way since, unlike the class `IldRequest`, the class `IldDbms` has no `getStatus` member function. You cannot use this function to perform `select` statements because you cannot get the private `IldRequest` object that was used by the `IldDbms` object.

Through the class `IldSQLType`, IBM ILOG DB Link offers a full interface to find out the database type names to be used when creating a table. The application can retrieve the

proper RDBMS-dependent name for a column using the `IldDbms::getTypeInfo` function, which takes its first argument from the list at the end of the `ildconst.h` file. This function can return several objects in an array, or no objects if that specific type does not exist in the currently connected RDBMS.

Transaction Control

Most RDBMSs can handle sequences of SQL statements as one block: all statements succeed or all fail. This typical behavior is called a *transaction mechanism*.

In some rare cases, the RDBMS is not capable of handling transactions, because this functionality either is not implemented or not enabled for the database, as with Informix.

A block is delimited by the *initiation* of the transaction and by its *commitment* or *rollback*.

The activation of the transaction mechanism can be checked by a call to one of the functions `IldDbms::isTransactionEnabled` or `IldDbms::getInfo(IldTransactionCapable,...)`.

IBM ILOG DB Link offers a unified API that avoids RDBMS specificity. The member functions in the API take two optional arguments. However, for portability considerations, we urge you to always pass a value to each optional argument and to stick to the protocol that consists of sending the SQL statements by means of the `IldRequest` object that initiated the transaction. If you do not follow this rule when using MS SQL Server, you will encounter an unexpected behavior: the request that initiates the transaction is not the one that executes the SQL statements. This amounts to an empty transaction and results in the following:

- If the transaction is rolled back, the changes to the database will not be undone.
- It the transaction is committed, the changes are executed but they are not validated. Therefore, they are lost on disconnection.

With IBM ILOG DB Link, you can use member functions of the class `IldDbms` for any transaction-related command, whatever your target RDBMS, as shown in Table 3.5:

Table 3.5 *IldDbms Member Functions for Transaction-Related Commands*

To	Use the `IldDms` member function	Comments
Initiate a transaction	`IldDbms::startTransaction`	Inoperative with Oracle and ODBC ports
Commit a transaction	`IldDbms::commit`	

Table 3.5 *IldDbms Member Functions for Transaction-Related Commands*

To	Use the `IldDms` member function	Comments
Roll back a transaction	`IldDbms::rollback`	
Switch the auto-commit mode on or off	`IldDbms::autoCommitOn` or `IldDbms::autoCommitOff`	Inoperative with Sybase and MS SQL Server.

For all these functions, the `request` argument (a pointer to an `IldRequest` object) is optional for most of the supported RDBMSs. However, this argument is mandatory for the Sybase and MS SQL Server database systems, which all require the `IldRequest` object to control the commands that execute the SQL statements enclosed in the transaction.

Some RDBMSs have implemented the “auto-commit mode” feature. When on, this mode commits each SQL statement when it is executed. Each database system has a unique notion of transaction control and, therefore, a unique interface to implement it.

With Sybase and MS SQL Server, the first argument is mandatory for all transaction-control functions. The second argument is used only for Sybase. Sybase TransactSQL allows you to name a transaction. For all other ports, both arguments are ignored. The following items relevant to transactions are described:

- *Initiating a Transaction*
- *Committing a Transaction*
- *Rolling Back a Transaction*
- *Autocommit Mode*

Initiating a Transaction

To initiate a transaction, call the member function `IldDbms::startTransaction`, as shown in the following example:

```
{
   // A new transaction.
   cout << "Initiating a transaction." << endl;
   if (!dbms->startTransaction(request))
      IldDisplayError("Begin transaction failed:", dbms);
}
```

This function takes two arguments, which are optional for most RDBMSs.

- If specified, the first argument must be a pointer to an `IldRequest` object, which is used to send the SQL statements that make up the body of the transaction.
- If specified, the second argument is a character string that is set as the transaction name.

With Informix, it is an error to start, commit, or roll back a transaction several times on a connection. IBM ILOG DB Link ensures that superfluous calls to these functions will not raise an error: they simply do nothing. However, you must be aware that the actual transaction still starts with the first call to `IldDbms::startTransaction` and ends with the first call to either `IldDbms::commit` or `IldDbms::rollback`.

Committing a Transaction

To commit a transaction, call the member function `IldDbms::commit`. This function takes two optional arguments, as shown in the following example.

```
{
   cout << "Committing the transaction." << endl;
   if (!dbms->commit(request))
      IldDisplayError("Commit failed: ", dbms);
}
```

- If specified, the first optional argument must be a pointer to the `IldRequest` object used to send the SQL statements that make up the body of the transaction—namely, the same object that was used to initiate the transaction.
- If specified, the second optional argument is the same transaction name that was used to initiate the transaction.

Rolling Back a Transaction

To roll back all effects of the SQL statements executed since the transaction was initiated, call the member function `IldDbms::rollback`, as shown in the following example:

```
{
   cout << "Rolling back the transaction." << endl;
   if (!dbms->rollback(request))
      IldDisplayError("Rollback failed:", dbms);
}
```

This member function takes two optional arguments, which must be the same as those used to initiate the transaction.

Autocommit Mode

To switch the autocommit mode on or off, use the member function `IldDbms::autoCommitOn` or `IldDbms::autoCommitOff`. Like other transaction-control functions, these functions take two optional arguments.

> ***Note:*** *The autocommit mode, while ensuring commitment of every successful SQL statement, exacts a very high price in terms of server work. You should avoid setting it on when it is not required by the application context.*

Cursor Allocation

A cursor is an instance of the class `IldRequest`. Although this name is currently used in IBM ILOG DB Link manuals, it is not fully appropriate: an `IldRequest` object is actually used to handle any SQL statement whether it needs a cursor or not.

The function `IldDbms::getFreeRequest` registers a newly created cursor in its cursor array. When a cursor is deleted—either explicitly using the `delete` operator or implicitly on disconnection—the corresponding connection is notified of the cursor disappearance.

> ***Important:*** `IldRequestModel` *derived objectsdo not behave in this way because, by defaut, they are not attached to any* `IldDbmsModel` *object.*

Extending the IldDbms Class

It is not possible to derive from the class `IldDbms` because the actual objects handled by your application are instances of its subclasses rather than instances of the base class.

If, for any reason, you need to extend the functionality of the `IldDbms` class, you can retrieve the determining part for connection handling. The member function `IldDbms::getHook` returns that part as a void pointer.

Table 3.6 shows what this function returns and what proprietary client interface you can call, depending on the target database system.

Table 3.6 *Values Returned by the Member Function IldDbms::getHook*

With this Database System...	**...the `getHook` function returns**	**...and lets you call the following database proprietary client interface.**
DB2	the SQLHANDLE	CLI (Call Level Interface)
Informix	the connection name (a character string)	Embedded SQL
MS SQL Server	the pointer to the DBPROCESS structure	DB Library function
ODBC	the `HDBC`	ODBC functions
OLE-DB	the pointer to the IDBCreateSession structure	OLE-DB functions
Oracle9, 10 or 11	the pointer to the OCISvcCtx structure	OCI functions
Sybase	the pointer to the `CS_CONNECTION` structure	Client Library functions

On the other hand, the member function `IldDbms::setHook` may be used to initialize the connection with an existing connection from an other application.

The `IldDbms` instance must be disconnected before using this function.

The connection given will not be closed when calling `IldDbms::~IldDbms()`, or when calling `IldDbms::disconnect()`. Since it was allocated outside of IBM ILOG DB Link, the application is responsible for closing it.

Use Notification

The notification is implemented in the ‘Enterprise’ or ‘Workstation’ editions of Oracle.

The following entities are required:

- a queue (either persistent or not),
- a subscriber attached to this queue,
- a trigger for the event.

The trigger will send a message to the queue, and the application will be notified that an event was generated.

This feature lets you receive notifications from the RDBMS asynchronously. This means that the application may be performing any task when the notification is received. A handler is automatically called when the notification is received. When it is completed, the execution will continue the interrupted task.

Subscribe to an Event

Subscription to an event is done using the following function:

```
IldDbms::subscribeEvent(const char* name,
                        IldNotifFunction usrCB,
                        IlAny usrData) ;
```

The name given will be <queue name>:<agent name>.

- `usrCB` is the handle that will be called when the event is received.
- `usrData` is a buffer from the application that will be transmitted to the callback function.

Unsubscribe from an Event

Unsubscribing is done using the following function:

```
IldDbms::unSubscribeEvent(const char* name) ;
```

- `name` is the name that identifies the event (the one used to subscribe to the event).

This function is to be called when you no longer need to receive notifications for a given event.

Differences between IldDbms and IldDbmsModel Classes

`IldDbmsModel` provides the same functionalities as `IldDbms`, plus the ability to be derived. This derivation capability introduces a few differences with the `IldDbms` class, as follows:

- `IldDbmsModel` instances are allocated directly by their constructor. The schema of using a function such as `IldNewDbms()` is not used for the `IldDbmsModel` class.
- The `IldDbms::getFreeRequest` member function cannot be used to instantiate a request with an object of the `IldDbmsModel` class. Instead, you must use the `IldRequestModel::IldRequestModel` constructor, with the `IldDbmsModel` instance as a parameter.
- When an `IldDbms` instance is deleted, the `IldRequest` instances that are linked to it are automatically deleted. This functionality is not implemented for the `IldDbmsModel` class. You must delete all `IldRequestModel` instances allocated using the given `IldDbmsModel` instance.

  (This is because `IldRequestModel` instances may be allocated on the stack, as automatic instances. Therefore, IBM ILOG DB Link has no control over the deletion of those instances. They will be deleted when the block in which they have been allocated ends.)

C H A P T E R

4

Cursors

IBM® ILOG® DB Link implements cursors as instances of the class `IldRequest` or `IldRequestModel`. `IldRequestModel` is the twin class to `IldRequest`, but in addition it provides the ability to be derived. In this manual we describe the `IldRequest` class, which is the main class. All its features also apply to the `IldRequestModel` class. The few differences between the two twin classes are itemized in a subsection.

This chapter is divided as follows:

- *IldRequest Objects* — Creating, manipulating, and deleting `IldRequest` objects.
- *Configuration Settings* — Default settings, accessing and changing the configuration, and array modes.
- *Column and Parameter Descriptors* — The concept of descriptors, implementation descriptors, and application descriptors.
- *Processing SQL Statements* — Immediate execution, deferred execution, and preparing a statement.
- *Results Retrieval* — Fetching and handling result sets.
- *Binding Input Variables* — How to use the `IldRequest::bindParam` function.
- *Generic Data Types* — Handling date, time, and numeric values.
- *Large Objects (LOBs)* — Sending and retrieving large objects.

- *Handling Abstract Data Type Values* — Using the descriptors of user-defined data types with ORDBMSs.
- *Extending the IldRequest Class* — This class cannot be derived but you can extend its functionalities under certain conditions. If you need to derive the `IldRequest` class, you must use its twin class, `IldRequestModel`.
- *Differences between IldRequest and IldRequestModel Classes*

IldRequest Objects

This section explains how to create, manipulate, and delete `IldRequest` objects. It is divided as follows:

- *Creating IldRequest Objects*
- *Number of Active Cursors*
- *Disposing of IldRequest Objects*

Creating IldRequest Objects

No public constructor exists for the class `IldRequest`. The `IldRequest` constructor is private to its class so it cannot be called from your application.

To create an `IldRequest` object, you must have already created an `IldDbms` object using the function `IldNewDbms`.

The only way to create a cursor is to ask an `IldDbms` object to deliver one using the function `IldDbms::getFreeRequest`, as shown in the following example:

```
{
   cout << "Creating a request: " << endl;
   IldRequest* request = dbms->getFreeRequest();
   if (dbms->isErrorRaised()) {
      IldDisplayError("Creation of request failed: ", dbms);
      delete dbms;
      exit(1);
}
```

The `IldDbms::getFreeRequest` function does not necessarily allocate a new `IldRequest` object each time it is called, but instead, it may reuse any `IldRequest` object

that has been previously released (see *Releasing an IldRequest Object* for more information).

> ***Warning:*** *It is possible for the allocation of an* `IldRequest` *object to succeed (partially, for example), while the creation of the object not succeed. In such a situation, memory was at least partially allocated, but the returned object cannot be used to execute a query. For that reason, you must always check that no errors were raised in the* `IldDbms` *object.*

If an error is raised but you do not check it, you will be using a special object instead of the normal `IldRequest` object. Using this special object will, in turn, raise an error `ILD_USING_ERROR_REQUEST` each time any function is called with it.

Number of Active Cursors

The number of `IldRequest` objects you can create is limited only by the database system configuration (for example, 50 with the Oracle default configuration).

To get the number of active `IldRequest` objects, you call the member function `IldDbms::getNumberOfRequests`. This number corresponds to the number of `IldRequest` objects that actually exist, not the number of `IldRequest` objects for which an SQL statement is being processed. That is, even if an `IldRequest` object has been released, it is still considered active (see next section for more information).

Disposing of IldRequest Objects

Disposing of an `IldRequest` object involves releasing it and destroying it.

Releasing an IldRequest Object

When you are finished using an `IldRequest` object, you can tell IBM ILOG DB Link that you are not going to use it any longer and that the object is at its disposal. To do so, use the member function `IldRequest::release`.

> ***Warning:*** *You must be careful not to use an* `IldRequest` *object once it has been released. Instead, you must ask the* `IldDbms` *object to supply a new* `IldRequest` *object (which could be the same).*

Destroying an IldRequest Object

An `IldRequest` object can be destroyed and its server-side allocated resources released on an explicit or implicit basis.

- To destroy an `IldRequest` object explicitly, just call the C++ operator `delete` on it. The `IldRequest::~IldRequest` destructor notifies the related `IldDbms` object of its disappearance.

- To destroy an `IldRequest` object implicitly, all you have to do is leave the object where it is. Actually, the destruction of the related `IldDbms` object causes the appropriate destructor to be called.

Releasing Versus Destroying

IBM ILOG DB Link tries to manage the memory allocated for `IldRequest` objects as sparingly as possible. This is why you are strongly advised to use the function `IldRequest::release` rather than calling the operator `delete` when it comes to disposing of an `IldRequest` object.

Using the pair `IldDbms::getFreeRequest`/`IldRequest::release` is, on average, faster than using the pair `IldDbms::getFreeRequest`/`delete` because, with the first pair, the `IldRequest` object is not deleted and will be reused on a further call to `IldDbms::getFreeRequest`.

Configuration Settings

This section describes how to access and change configuration settings and tells you more about the array bind and array fetch modes. It is divided as follows:

- *Default Settings*
- *Accessing and Changing the Configuration*
- *Array Modes*

Default Settings

Any `IldRequest` object returned by the function `IldDbms::getFreeRequest` is configured using the current configuration setting from its related `IldDbms` object.

Thus, it inherits the array fetch size and the parameter array size, as well as the settings for the *date as string*, *numeric as string*, and *numeric as object* features.

> ***Note:*** *The error reporter is not reset when an* `IldRequest` *object has been released and is later reassigned by* `IldDbms::getFreeRequest`.

Accessing and Changing the Configuration

Table 4.1 shows what member functions of the class `IldRequest` you can use to change the default configuration settings.

Table 4.1 *Changing the Default Configuration Settings*

Use	To
`IldRequest::setColArraySize`	Change the fetch array size
`IldRequest::removeColArraySize`	Return the cursor to the default fetching protocol (one row at a time)
`IldRequest::getColArraySize`	Retrieve the current fetch array size
`IldRequest::setParamArraySize`	Set the parameter array size
`IldRequest::removeParamArraySize`	Return the cursor to the default binding protocol (one parameter row at a time)
`IldRequest::getParamArraySize`	Retrieve the current parameter array size

Array Modes

IBM ILOG DB Link can handle several rows at a time, whether input or output data. The default setting, however, is one row at a time.

Each `IldRequest` object inherits the settings of its related `IldDbms` object. The default settings can be changed at the `IldDbms` level.

While using array modes enhances performance —at the network communication level for *array fetch* and with respect to CPU time for *array bind*—you must be aware that IBM ILOG DB Link pre-allocates memory for data values and null indicators. The data buffers are allocated the maximum size required for the column data types, except for the LOB types `IldLongTextType` and `IldBinaryType`, for which the buffer size is limited to 64 Kilobytes. Consequently, on some systems with limited memory, setting the array mode to a high number of rows may cause an allocation failure.

Array Bind Mode

- To set the *array bind* mode, specify the number of rows you want to send at a time. This number is a maximum and can be changed by the value of the second argument passed to the member function `IldRequest::execute`.

For each `IldRequest` object, you can change the array size using the function `IldRequest::setParamArraySize` with a positive integer as its argument:

```
{
   cout << "Host variables array size set to 2" << endl;
   request->setParamArraySize(2);
}
```

- To get the array size, use the member function `IldRequest::getParamArraySize`.
- To reset the array size, use the function `IldRequest::removeParamArraySize`.

Warning: *To be effective, the array bind size must be set before the function* `parse` *or* `execute` *is called.*

Array Fetch Mode

Since the cursor-relative positioning and absolute positioning are not implemented, these features do not prevent you from using the function `IldRequest::fetch`.

- To set the default value for all newly created `IldRequest` objects, use the function `IldDbms::setDefaultColArraySize` with a positive integer as its argument.

```
{
   dbms->setDefaultColArraySize((IlUInt)10);
}
```

- To change this setting for a given `IldRequest` object, use the member function `IldRequest::setColArraySize` with a positive integer as its argument.
- To reset this setting, use the function `IldRequest::removeColArraySize`.

Warning: *To be effective, the array fetch size must be set before the first call to the* `fetch` *member function.*

- To get the current array size, use the function `IldRequest::getColArraySize`.

With ODBC, the array fetch mode is available only if the driver has level-2 compliance.

Column and Parameter Descriptors

This section presents application and implementation descriptors as implemented by IBM ILOG DB Link. It is divided as follows:

- *Notion of Descriptors*
- *Implementation Descriptors*
- *Application Descriptors*

Notion of Descriptors

CLI Definition

The CLI standard defines descriptors at implementation- and application- levels.

- At implementation level —that is, from the database server point of view— the descriptors are called IRD (Implementation Row Descriptor) and IPD (Implementation Parameter Descriptor).
- At application level, the descriptors are called ARD (Application Row Descriptor) and APD (Application Parameter Descriptor).

Likewise, IBM ILOG DB Link also differentiates between implementation level and application level, but refers to row descriptors as column descriptors and uses the same classes for column and parameter descriptors.

Implementation Within IBM ILOG DB Link

IBM ILOG DB Link uses two classes to describe the properties of a column or parameter:

- `IldDescriptor`: Declared in the file `ildent.h`, this class is used to describe a column or parameter type on the server side.
- `IldAppDescriptor`: A subclass of `IldDescriptor` declared in the file `ildtuple.h`, this class is used to hold the type descriptor and the column or parameter values and indicators on the application side.

Implementation Descriptors

An instance of `IldDescriptor` holds the following information about the column or parameter. Table 4.2 shows the corresponding member functions:

Table 4.2 *IldDescriptor Member Functions*

Use this Member Function	To get
`IldDescriptor::getType`	IBM ILOG DB Link type
`IldDescriptor::getSqlType`	SQL data type code
`IldDescriptor::getName`	Column name
`IldDescriptor::getSize`	Maximum data size (in bytes)
`IldDescriptor::getPrecision`	Precision (for columns of a numeric type) - `0` if irrelevant
`IldDescriptor::getScale`	Scale (for columns of a numeric type) - `0` if irrelevant
`IldDescriptor::getSqlTypeName`	SQL type name in the server
`IldDescriptor::isNullable`	Nullability flag
`IldDescriptor::getADTDescriptor`	Abstract type descriptor (for columns of IBM ILOG DB Link type) - `null` otherwise

Note: *Since most RDBMSs do not have the capability of describing the parameters of a query, the contents of the* `IldDescriptor` *object for a parameter are undefined until it is bound using the member function* `IldRequest::bindParam`.

Type Codes

SQL data type codes are defined as constants in the section `Type Codes` of the `ildconst.h` header file. Most of them have a name that is very similar to their CLI name but a few of them differ. The names are prefixed with `IldSQL`.

Their values are the ones defined in the CLI standard.

The CLI specification has been extended to negative values so as to provide support for all types of all supported RDBMSs.

Application Descriptors

An instance of `IldAppDescriptor` adds the following information to its base class `IldDescriptor`:

Table 4.3 *Member functions added by IldAppDescriptor to IldDescriptor*

Information	Member Function
Size of one element in the value buffer (in bytes)	`IldAppDescriptor::getBufferSize`
Value buffer	`IldAppDescriptor::getValue`
Indicator buffer	`IldAppDescriptor::getNulls`

In addition:

- The member function `IldAppDescriptor::isExtValue` lets you know whether the value buffer was allocated by IBM ILOG DB Link or is bound to some application memory space. This function returns `IlFalse` in the first case or `IlTrue` in the second.
- The member function `IldAppDescriptor::isExtNulls` lets you know whether the null-indicator buffer was allocated by IBM ILOG DB Link or is bound to some application memory space. This function returns `IlFalse` in the first case or `IlTrue` in the second.

Simple `IldDescriptor` objects exist only for user-defined data-type attribute descriptors of type `IldADTDescriptor`. All other accesses to descriptors return instances of derived classes:

- The function `IldRelation::getColumn` returns instances of the derived class `IldColumn`.
- The functions `IldRequest::getColDescriptor` and `IldRequest::getParamDescriptor` return instances of the derived class `IldAppDescriptor`.

Processing SQL Statements

SQL statement are usually sent for processing one at a time. However, some RDBMSs allow you to send a batch of several statements at once: such is the case of Sybase, MS SQL Server, and ODBC if the underlying RDBMS allows it.

Although it is not forbidden by IBM ILOG DB Link, we do not recommend using batches of statements. Stored procedures are far more convenient.

Two different processes exist with respect to statement execution:

- ***Immediate Execution***: The SQL statement is sent to the server immediately via the function `IldRequest::execute`.
- ***Deferred Execution***: The execution process is broken down into the following steps:
  - The SQL statement is prepared by the function `IldRequest::parse`.
  - The parameters are bound and set.
  - The actual execution takes place when the function `IldRequest::execute` is called.

You should choose immediate execution when the query has no placeholder (or parameter) and will be used only once.

Immediate Execution

To execute an SQL statement immediately, use the member function `IldRequest::execute`, which takes two arguments:

- The first argument is the SQL statement string.
- The second argument, `rowCount`, is optional. If specified, it must be a valid pointer to an `IlInt` variable, which will be set to the number of processed rows if the statement is `delete`, `insert`, or `update`.

> ***Note:*** *rowCount is zero after a select query is executed. For performance reasons, most RDBMSs do not "look ahead" to set this value for a query.*

Once the call to `IldRequest::execute` succeeds, you can retrieve the execution status via the member function `IldRequest::getStatus`. This member function returns the number of processed rows if the SQL statement is `delete`, `insert`, or `update`. If it is a

`select` statement, the returned value is the number of rows actually fetched —namely 0 when the execution has just been completed.

> ***Note:*** *An SQL statement that is to be executed immediately must not contain any parameters.*

If the execution fails, the returned value, usually -1, is not meaningful.

```
{
   if (!request->execute(queryBuffer))
      IldDisplayError("Executing: ", request);
   else {
      if (!request->fetch())
         IldDisplayError("Fetching: ", request);
      else if (request->hasTuple())
   // Loop with two levels, since ODBC, Sybase, and MS SQL Server
   // can have several result sets for one command.
      do {
         if (request->getColCount()) {
            IldPrintTuple(request, IldNames);
            IldPrintTuple(request, IldSeparators);
            IldPrintTuple(request);
            while (request->fetch().hasTuple())
               IldPrintTuple(request);
            cout << endl;
            if (request->isErrorRaised())
               IldDisplayError("Fetching: ", request);
      }
      } while (request->fetch().hasTuple());
   else if (request->getColCount()) {
      IldPrintTuple(request, IldNames);
      IldPrintTuple(request, IldSeparators);
      cout << endl << "No row found." << endl;
   }
   else if ((count = request->getStatus()) > 0)
      cout << count << " modified line(s)" << endl;
   }
}
```

It is also possible to get the attributes of the returned rows from the result set, using the following member functions:

- `IldRequest::getColName` to get the column names,
- `IldRequest::getColType` to get the column types,
- `IldRequest::getColSize` to get the column sizes.

If the SQL statement is `select`, you can retrieve the rows using the member function `IldRequest::fetch`. When used after execution of any other SQL statement, this member function is ineffective, but no error is raised.

Deferred Execution

It is not always useful to execute a query immediately, either because it contains placeholders for which values must be passed or because you want to reuse the same query and thus avoid the time needed to prepare it for execution.

The database system must prepare a query before executing it. Preparing a query involves:

1. Parsing the query and checking the SQL syntax,
2. Preparing an execution plan,
3. Calling the query optimizer.

These steps can take place only once, rather than repeatedly, if you intend to use the same query several times —whether containing placeholder values or not.

To send the same query several times, you must follow this protocol:

1. Prepare the query by calling `IldRequest::parse`.
2. Bind the placeholders by calling `IldRequest::bindParam`.
3. If needed, bind output columns by calling `IldRequest::bindCol`.
4. For each execution, pass values to the placeholders by calling the function `IldRequest::setParamValue`.
5. Call `IldRequest::execute`.

> ***Note:*** *When using any of the member functions related to the result set or to the parameters set, remember that IBM ILOG DB Link follows the C or C++ convention, where indexes start from 0, rather than the standard SQL convention, where indexes start at 1.*

Here is an example:

```
{
   const char* selectStr = (!strcmp(dbms->getName(), "oracle") ||
                            !strcmp(dbms->getName(), "sqlbase")) ?
   "select * from DBLTABLE where no > :1"
   :
   "select * from DBLTABLE where no > ?";
    cout << "Parsing a request containing a host variable: " << endl;
   cout << "\t" << selectStr << endl;
   if (!request->parse(selectStr)) {
      IldDisplayError("Parse failed: ", request);
      Ending(dbms, request);
   }
   cout << endl;
    cout << "Host variable bind with integer type and value 0" << endl;
   if (!request->bindParam((IlUShort)0, IldIntegerType)) {
      IldDisplayError("Binding failed: ", request);
```

```
      Ending(dbms, request);
   }
   request->setParamValue((IlInt)0, 0);
      cout << "Executing the request" << endl;
   if (!request->execute()) {
      IldDisplayError("Execute failed: ", request);
      Ending(dbms, request);
   }
   cout << endl;
      cout << "Results from the request: " << endl;
   if (request->getColCount()) {
      while (request->fetch().hasTuple()) {
         if (!request->isColNull(0))
            cout << request->getColIntegerValue(0);
         else
            cout << "-";
         cout << "\t";
         if (!request->isColNull(1))
            cout << request->getColStringValue(1);
         else
            cout << "-";
         cout << endl;
      }
   }
}
```

Preparing a Statement

To prepare a statement, issue a call to `IldRequest::parse`. Its argument is the SQL statement string to be prepared. Its action is roughly equivalent to the SQL statements `PREPARE` and `DESCRIBE`.

Warning: *Parsing a request discards any previously parsed queries, as well as all pending result sets of the* `IldRequest` *object.*

This `parse` method must not be called for statements that cannot be prepared. These statements vary depending on the RDBMS you are connected to. Check the RDBMS client API manuals for information about which statements can be prepared.

Note: *A simple* `select` *statement that includes no placeholder should not be prepared when you use the ODBC port. The resulting data will be irrelevant in the bound memory space.*

Multiple Execution

Multiple execution means that one call to the member function `IldRequest::execute` will process several rows in the database. This function takes two optional arguments:

- The first argument is a pointer to an `IlInt` variable. After execution is successful, this argument is set to the number of processed rows.

- The second argument, a number of type `IlUInt`, is the number of times the query has to be executed. When specified, this number must be positive and less than or equal to the value returned by a call to the function `IldRequest::getParamArraySize`.

> ***Note:*** *When you use ODBC, this second argument can be set only if the driver is ODBC level 2 compliant. Also with ODBC, the array bind mode can be used only if the driver is ODBC level 2 compliant.*

```
{
   // Since the "count" argument is set to 1, there will be only one
   // update performed despite the variable array size set to 2 !!
   if (!request->execute(&rowCount, 1)) {
      IldDisplayError("Execution failed: ", request);
      Ending(dbms, request);
   cout << "Row processed count " << rowCount << endl << endl;
}
```

Repeated Execution

Once prepared, a query can be executed as many times as needed by successive calls to the overloaded member function `IldRequest::execute`. Before each execution, you can set new bindings for the input variables (placeholders) and the output columns. You must also pass the values of the input variables.

Example:

```
{
   for (int i = 0; i < 5; i++) {
      cout << "Set " << i << "th variable to name: "
           << names[i] << " and age: " << ages[i] << endl;
      request->setParamValue(names[i], 0, 0);
      request->setParamValue(ages[i], 1, 0);

      //  execute the insertion
      cout << "Inserting row" << endl;
      if (!request->execute(&n)) {
         IldDisplayError("Execution failed: ", request);
         delete cust;
         Ending(dbms, request);
      }
      cout << n << "row(s) inserted." << endl;
   }
}
```

A frequent mistake is to use the basic member function `IldRequest::execute`. This function takes a string as its first argument, thus causing an error, in this context, due to unbound variables.

Results Retrieval

Once an SQL `select` statement has been executed as a query, the resulting data can be fetched using the member function `IldRequest::fetch` or `IldRequest::fetchScroll`.

The member function `IldRequest::fetchScroll` is implemented for RDBMSs that enable this feature: Informix®, Mssql, Oracle® and Odbc.

You can go back in the result set when you use this function, while `IldRequest::fetch` lets you only access the records that follow it.

To activate this feature, you must activate the scrollable cursor mode by calling `IldRequest::setScrollable(IlTrue)`. This feature is not activated by default because it requires additional resources on the server side. Furthermore, the RDBMS may restrict the set of queries you can run. For instance, with Informix, you cannot fetch a BYTE or TEXT column using a scrollable cursor.

When the column array size is used to fetch several rows at the same time, the <n> last rows will be retrieved if `fetchOrientation == IldFetchDirectionLast`.

Once fetched, the data can be read using the specialized accessors `IldRequest::getCol<type>Value`, where `<type>` is one of the valid IBM ILOG DB Link column type names (see *Direct Access* for details).

A faster way to retrieve and send data is to bind application-allocated memory to IBM ILOG DB Link. This can be done for input values as well as for output columns.

The following items are described individually:

- *Handling Multiple Result Sets*
- *Direct Access*
- *Binding to User-Allocated Memory*

Handling Multiple Result Sets

A *result set* is returned at fetch time for each SQL `select` statement. Sybase and MS SQL Server return several result sets for a stored procedure call.

When the target RDBMS allows queries to be sent in batches, there may exist several successive result sets to be fetched. In this case, IBM ILOG DB Link retrieves the first set, then returns the value `IlFalse` from the call to `IldRequest::hasTuple`.

To make sure that there is no other result set to be fetched, you must issue another call to `IldRequest::fetch` and then a call to `IldRequest::hasTuple`, as shown in the following example:

```
{
```

```
   cout << "\tResult sets: " << endl;
   while (request->fetch().hasTuple()) {
      if (request->getColCount()) {
         IldPrintTuple(request, IldNames);
         IldPrintTuple(request, IldSeparators);
         do {
            IldPrintTuple(request);
         } while (request->fetch().hasTuple());
   }
   else {
      IlInt count = 0;
      if ((count = request->getStatus()) > 0)
         cout << count << " modified row"
              << ((count > 1) ? "s" : "") << endl;
      }
      cout << endl;
   }
}
```

Direct Access

To retrieve data, you can either:

- use the IBM ILOG DB Link API alone with type-related member functions,
- or bind the application memory space on output.

IBM ILOG DB Link data accessors are of the form `IldRequest::getCol<type>Value` where `<type>` can be one of the following IBM ILOG DB Link types:

- `ADT`
- `Binary`
- `Byte`
- `Date`
- `DateTime`
- `Integer`
- `LongText`
- `Money`
- `Numeric`
- `Real`
- `Ref`
- `String`

Here is an example:

```
{
   // Selection of the data accessor.
   if (request->isColNull(i))
      ItemsArray[i]._buffer = "-";
```

```
   else
      switch (request->getColType(i)) {
      case IldDateType:
         ItemsArray[i]._buffer =
            IldStrRTrim((char*)request->getColDateValue(i));
         break;
      case IldDateTimeType:
         ItemsArray[i]._buffer =
            IldDateTimeToString(request->getColDateTimeValue(i));
         break;
      case IldStringType:
         ItemsArray[i]._mode = IldLeft;
         ItemsArray[i]._buffer =
            (char*)request->getPurgedStringValue(i);
         break;
      case IldLongTextType:
         ItemsArray[i]._mode = IldLeft;
         ItemsArray[i]._buffer =
            IldStrRTrim((char*)request->getLongTextValue(i));
         break;
      case IldMoneyType: {
         ItemsArray[i]._buffer = &BuffersArray[i * IldBufSize];
         ostrstream ostr(&BuffersArray[i * IldBufSize],
                    (int)IldBufSize);
         ostr << '$' << request->getMoneyValue(i) << ends;
         break;
      }
      case IldRealType: {
         ItemsArray[i]._buffer = &BuffersArray[i * IldBufSize];
         ostrstream ostr(&BuffersArray[i * IldBufSize],
                    (int)IldBufSize);
         ostr << request->getRealValue(i) << ends;
         break;
      }
      case IldByteType: {
         ItemsArray[i]._buffer = &BuffersArray[i * IldBufSize];
         ostrstream ostr(&BuffersArray[i * IldBufSize],
                         IldBufSize);
         ostr << (short)request->getByteValue(i) << ends;
         break;
      }
      case IldIntegerType: {
         ItemsArray[i]._buffer = &BuffersArray[i * IldBufSize];
         ostrstream ostr(&BuffersArray[i * IldBufSize],
                        IldBufSize);
         ostr << request->getIntegerValue(i) << ends;
         break;
      }
      case IldBinaryType: {
         ItemsArray[i]._buffer = "...";
         break;
      }
      case IldUnknownType: {
         ItemsArray[i]._buffer = "???";
         break;
      }
   }
}
```

Also, you should keep in mind that:

- When the value is of type `IldStringType`, `IldBinaryType`, or `IldLongTextType`, it must be copied over to the memory allocated by your application because after the next call to `IldRequest::fetch`, the value will be undetermined.
- Values returned for columns of type `IldDateTimeType` or `IldNumericType` are automatically copied into the receiving object.
- Values returned for columns of type `IldObjectType` or `IldCollectionType` become the property of the application. Therefore, the application must delete them when they are no longer needed.

Binding to User-Allocated Memory

An interesting optimization consists of retrieving the data directly into your application memory space. To achieve this, use one of the overloaded member functions `IldRequest::bindCol`. The first one uses the *column index* in the result set as a key, whereas the second one uses the *column name*.

If you choose this method, be aware that when IBM ILOG DB Link compares the strings, the comparison is case-sensitive. Some RDBMSs use only uppercase while others use lowercase letters, or are case-sensitive themselves.

To find out how your target RDBMS handles character cases, you can call the function `IldDbms::getInfo` for information item `IldIdentifierCase`.

Here is an example:

```
{
   //   1/ parse a selection request
   const char* selectStr = "select NAME, AGE from BINDTABLE";
   cout << "Parsing select request: " << selectStr << endl;
   if (!request->parse(selectStr)) {
      IldDisplayError("Parse failed: ", request);
      delete cust;
      Ending(dbms, request);
   }
   cout << endl;
   //   2/ declare binding of outputs
   cout << "Binding output: " << endl;
   cout << "  column NAME bound to customer name slot" <<   endl;
   if (!request->bindCol((IlUShort)0, IldStringType,
      cust->name, 20)) {
      IldDisplayError("Column binding failed:", request);
      delete cust;
      Ending(dbms, request);
   }
   cout << "  column AGE bound to customer age slot" << endl;
   if (!request->bindCol(1, IldIntegerType, &(cust->age))) {
      IldDisplayError("Column binding failed:", request);
      delete cust;
      Ending(dbms, request);
```

```
   }
   cout << endl;
   //   3/ execute the select request
   cout << "Executing the select request" << endl;
   if (!request->execute()) {
      IldDisplayError("Execution failed: ", request);
      delete cust;
      Ending(dbms, request);
   }
}
```

Binding Input Variables

To bind input variables, you will use the member function `IldRequest::bindParam`.

This section differentiates the standard implementation of this function from its overloaded versions, and explains how to set parameter values. It is divided as follows:

- *Standard Implementation*
- *Overloaded Version*
- *Setting Parameter Values*
- *Specific Considerations*

Standard Implementation

This function can take up to eight arguments, from which only the first two ones are mandatory, the others being optional:

- The first argument is the chronological order of the variable in the statement, that is, in a left-to-right reading, its order of appearance in the statement string.
- The second argument is the IBM ILOG DB Link type of the data.
- The third argument is the data size in bytes. It is ignored for fixed-size types (such as `integer`, `real`, `byte`) but it can be used for all other type bindings. However, the error `ILD_BAD_VARIABLE_SIZE` is raised if the size passed is too small to handle the values.
- The fourth and fifth arguments are, respectively, a pointer to the data and a pointer to the null indicator.
- The sixth argument indicates the variable output status. It is needed only with Sybase on stored procedure calls.

  The following code extract is taken from the sample file `sybproc.cpp` and shows how this argument is used:

```
{
   cout << "Binding parameters: " << endl;
   cout << " @intparam" << endl;
   if (!request->bindParam("@intparam", IldIntegerType)) {
      IldDisplayError("Binding of @intparam failed: ", request);
      delete dbms;
      return 1;
   }
   cout << " @sintparam" << endl;
   if (!request->bindParam("@sintparam", IldIntegerType,
                           -1, 0, 0, IlTrue)) {
      IldDisplayError("Binding of @sintparam failed: ", request);
      delete dbms;
      return 1;
   }
```

```
   cout << " @floatparam" << endl;
   if (!request->bindParam("@floatparam", IldRealType,
                           -1, 0, 0, IlTrue)) {
      IldDisplayError("Binding of @floatparam failed: ", request);
      delete dbms;
      return 1;
   }
   cout << " @charparam" << endl;
   if (!request->bindParam("@charparam", IldStringType, 20,
                           0, 0, IlTrue)) {
      IldDisplayError("Binding of @charparam failed: ", request);
      delete dbms;
      return 1;
   }
}
```

Warning: *IBM ILOG DB Link manages the memory allocation for data buffers internally. In case of long strings, as with Oracle* `VARCHAR` *or Informix* `CHAR`*, the allocated buffer may be too small if the size is not supplied at binding time. If your parameter is longer than 255 characters, pass the actual maximum size as the third argument to the function* `IldRequest::bindParam`*.*

- The seventh argument is the actual number of values in the array argument. This argument is only used with Oracle for stored procedure calls where array arguments are required.
- The eighth argument is a valid abstract data type descriptor. It is only used with ORDBMSs and is mandatory if the parameter is bound to `IldCollectionType` or `IldObjectType` types.

Warning: *Despite standardization, Oracle does not support the question mark as a variable identifier.*

Overloaded Version

There is an overloaded version of the function `IldRequest::bindParam` that is reserved for use with Oracle. Its first argument is a character string holding the parameter name the way Oracle expects it, that is, in the format `':'<parameter name>`.

Since Oracle also supports the format `':'<parameter number>`, the first implementation of the `bindParam` function can also be used with them, as shown in the following RDBMS example `sbinding`:

```
{
   cout << "Binding input variable :name of type string" << endl;
   if (!request->bindParam((IlUShort)0, IldStringType, 20)) {
      IldDisplayError("Variable binding failed:", request);
      delete cust;
      Ending(dbms, request);
   }
   cout << "Binding input variable :age of type integer" << endl;
   if (!request->bindParam(1, IldIntegerType)) {
      IldDisplayError("Variable binding failed:", request);
      delete cust;
      Ending(dbms, request);
   }
}
```

Setting Parameter Values

Just before calling the function `IldRequest::execute`, you must supply values for the parameters through the function `IldRequest::setParamValue`, as shown in the following example:

```
{
   IlInt n = 0;
   for (int i = 0; i < 5; i++) {
      cout << "Set " << i << "th variable to name: "
           << names[i] << " and age: " << ages[i] << endl;
      request->setParamValue(names[i], 0, 0);
      request->setParamValue(ages[i], 1, 0);

      //  execute the insertion
      cout << "Inserting row" << endl;
      if (!request->execute(&n)) {
         IldDisplayError("Execution failed: ", request);
         delete cust;
         Ending(dbms, request);
      }
      cout << n << "row(s) inserted." << endl;
   }
}
```

Specific Considerations

You must be careful when using variables of Oracle CHAR data type. For this data type, the values in the database are padded with blanks. A common error is to set parameter values for these columns, without padding the values with blanks.

For instance, with table article (name char(10), id char(10)), the query "select name from article where id = '0'" works correctly.

However, when calling parse for query "select name from article where id = :1", you have to fill the bind variable with spaces so that the equality operator retrieves the expected values.

Generic Data Types

Some database data types cannot be easily translated to C or C++ types. This section explains how IBM ILOG DB Link deals with this particular issue for:

- Handling Date and Time Values
- Handling Exact Numeric Values

Handling Date and Time Values

Date-and-time related data types are handled through different conversion protocols depending on the database systems. To achieve portability, IBM ILOG DB Link offers a feature that allows date-and-time values to be sent and retrieved as objects of the class `IldDateTime`.

The IBM ILOG DB Link class `IldDateTime` is a transparent container where you can put or get the separate components of a date-and-time value. Its fields extend the time precision to milliseconds.

> ***Note:*** *The millisecond precision of the class* `IldDateTime` *introduces a small discrepancy when connected to Informix, where a millionth of a second precision is possible.*

After turning the *date as string* feature off, you can use an `IldDateTime` object just like any other data type.

- To put a date-and-time value into a variable, use the member function `IldRequest::setParamValue`, like this:

```
{
    IldDateTime* dt = new IldDateTime(1996, 3, 2); // 1996/03/02
    request->setParamValue(dt, 1);
}
```

- To retrieve a date-and-time value, use the function `IldRequest::getColDateTimeValue`, as in the following code extract:

```
{
   // Print selected item values.
   do {
      if (!request->fetch())
         IldDisplayError("Fetch failed:", request);
      else if (request->hasTuple()) {
         cout << request->getColStringValue(0) << "\t";
         IldDateTime dt = request->getColDateTimeValue(1);
            if (request->isErrorRaised())
               IldDisplayError("Cannot retrieve DateTime: ", request);
            else {
               cout << dt.getYear() << "/" << dt.getMonth() << "/"
```

```
                    << dt.getDay() << " " << dt.getHour() << ":"
                         << dt.getMinute() << ":" << dt.getSecond() << endl;
            }
      }
   } while (request->hasTuple());
}
```

These two member functions raise an error of type `ILD_TYPE_MISMATCH` if the *date as string* feature is turned on.

At creation, all fields of an `IldDateTime` object are initialized by default at zero.

Handling Exact Numeric Values

Exact numeric types sometimes hold values that cannot be converted into integer values. They can be turned into floating-point values but with a loss in significant digits. IBM ILOG DB Link allows you to send or retrieve such values as character strings or objects to avoid that loss of precision.

Numeric As String

Very large numeric or decimal values can be handled as strings to preserve precision. To do so, turn on the *numeric as string* feature before you use the string-dedicated IBM ILOG DB Link member functions `IldRequest::setParamValue` and `IldRequest::getColStringValue`.

- To turn on the feature:

```
{
   // Data selection
   request->setStringNumericUse(IlTrue);
}
```

- To send huge exact numeric values as strings:

```
{
   request->setParamValue("9876543210987.654321098", 0);
}
```

- To retrieve such huge numeric values as strings:

```
{
   // Print selected item values.
   do {
      if (!request->fetch())
         IldDisplayError("Fetch failed:", request);
      else if (request->hasTuple()) {
         cout << request->getColStringValue(0) << "\t"
              << request->getColDateValue(1) << endl;
      }
   } while (request->hasTuple());
}
```

Numeric As Object

To gain full independence from the RDBMS server and client host localization, an application can use the *numeric as object* feature to handle values of data types `DECIMAL` and `NUMERIC`. The feature is enabled by a call to the function `IldIldBase::setNumericUse` with an argument value of `IlTrue`.

When using Oracle, due to the existence of the numeric type, `NUMBER`, numeric values are handled as objects as soon as that feature is turned on.

An `IlNumeric` object can be converted to and from the type `IlInt` but its value can also be accessed as a character string, which will be built following the `C` locale (no thousands separator and a dot as the decimal separator).

- To turn on the feature:

```
{
   // Use numeric objects
   request->setNumericUse(IlTrue);
}
```

- To send numeric object values:

```
{
   IlNumeric* num = new IlNumeric;
   num->set("987654321.654321");
   request->setParamValue(num, 0);
}
```

- To retrieve a numeric object value use the function `IldRequest::getColNumericValue`, like this:

```
{
   // Print selected item values.
   do {
      if (!request->fetch())
         IldDisplayError("Fetch failed:", request);
      else if (request->hasTuple()) {
         IlNumeric num = request->getColNumericValue(0);
         if (!request->isErrorRaised()) {
            char numValue[ILD_MAX_NUM_LEN];
            num.get(numValue, ILD_MAX_NUM_LEN);
            cout << numValue << "";
         }
      }
   } while (request->hasTuple());
}
```

Large Objects (LOBs)

With IBM ILOG DB Link, your application can handle Large OBjects (LOBs) either as a whole or in memory chunks.

LOBs can be sent to the RDBMS only from memory-stored data, but they can be retrieved as a whole into memory or as whole into a named file, or in chunks into application-allocated memory.

Retrieving LOBs in chunks depends on the RDBMS: most RDBMSs allow retrieval only in consecutive chunks. The only exception to this is Oracle whose API supports positioned retrieval.

The various RDBMSs use different type names for LOBs and have different protocols to send and retrieve these kinds of values. The term LOB is used for data types that allow values of unlimited size. (This is theoretical. In practice, the size is always limited to two gigabytes).

This section is divided in 2 parts:

- *Sending Large Objects*
- *Different Ways of Retrieving Large Objects*

Sending Large Objects

With IBM ILOG DB Link, you have only one way of sending LOBs to the database server, namely through the member functions `IldRequest::insertLongText` or `IldRequest::insertBinary`.

Both member functions actually *update* an already existing row using the discrimination clause passed in the fifth argument. This argument must contain a valid `where` clause, reduced to the predicate part —that is, without the `where` keyword.

Sending Text Data

The member function `IldRequest::insertLongText` takes five arguments:

- the text data buffer,
- the data length,
- a table name,
- a column name,
- a reduced `where` clause.

This code extract shows that you must send the column data as a whole.

```
{
   // Prepare the where clause of the update
   ostr.seekp(ios::beg);
   ostr << " NAME = '" << name << "'" << ends;
   // Find out file size
   int len = inFile.seekg(0, ios::end).tellg();
   char* buff = new char [len + 1];
   if (!buff) {
      cout << "Memory exhausted: cannot allocate text buffer" << endl;
      res = IlFalse;
   }
   else {
      // Read in data
      inFile.seekg(0, ios::beg); // Back to beginning of file
      inFile.read(buff, len);

      // Proper text insertion.
      if (!request->insertLongText(buff, len, "USERTABLE",
                                   "VALUE", str))
   res = IlFalse;
}
```

Sending Binary Data

The member function `IldRequest::insertBinary` takes four arguments:

- An `IldBytes` structure holding the data and its length,
- A table name,
- A column name, and
- A reduced `where` clause.

Different Ways of Retrieving Large Objects

To retrieve LOBs, you can choose one of the following three methods:

- Getting the whole contents at once into memory (with some limitations);
- Getting the whole contents at once into a file;
- Getting the contents in memory chunks.

Retrieving into Memory

- **Long Text Values**

To retrieve a large text value at once into memory, after having successfully executed an SQL `select` statement, use the function `IldRequest::getColLongTextValue`.

> ***Warning:*** *The memory chunk internally allocated by IBM ILOG DB Link is limited to 64 Kilobytes.*

```
{
   while (request->fetch().hasTuple())
      cout << request->getColLongTextValue(0) << endl;
}
```

- **Long Binary Values**

To retrieve a large binary value at once into memory, after having successfully executed an SQL `select` statement, use the member function `IldRequest::getColBinaryValue`.

> ***Warning:*** *The memory chunk internally allocated by IBM ILOG DB Link is limited to 64 Kilobytes.*

Retrieving into a Named File

To retrieve a large text value at once into a named file, use the member function `IldRequest::getLargeObject`. The member function itself issues the initial SQL `select` statement. There is no program limit on the size of the data value.

This function can be used for `IldLongTextType` values as well as for `IldBinaryType` values. It takes the following four arguments:

- The table name,
- The column name,
- A reduced `where` clause,
- The full path name of the file where the column data is to be saved.

```
{
   if (!request->getLargeObject("DBLTEXT", "TVAL",
                                        "NAME = '1st Text'","/tmp/text1")) {
      IldDisplayError("Text retrieval failed:", request);
      Ending(dbms, request);
   }
}
```

Retrieving in Chunks of Memory

To retrieve a large text or binary value in chunks of memory:

1. Initiate the process by a call to the function `IldRequest::startGetLargeObject`, which issues the initial SQL `select` statement.

   This function takes the following three arguments:

   - The table name,
   - The column name,
   - The reduced `where` clause.

2. Iterate by calling the function `IldRequest::getLargeObjectChunk` until the `offset` argument is left unchanged by the call, meaning that no more data was returned.

Since you pass the address of a preallocated memory chunk as the second argument to the call, it is up to you to decide the memory limitation.

The third argument is ignored on input for all RDBMSs except Oracle. Oracle allows you to fetch from any offset in the column value.

All other database systems only allow retrieving chunks by ascending, not by overlapping, indexes. On output, the argument is set to its entry value, increased by the total number of bytes actually read so far.

These functions can be used for `IldLongTextType` values as well as for `IldBinaryType` values.

Handling Abstract Data Type Values

Since *abstract data types* (also called *user-defined data types*) are supported only for ORDBMSs, the corresponding values make sense only when connected to databases that have this capability. Two such RDBMSs are currently supported: Oracle® and Informix® Universal Server, also called Informix9 in the IBM ILOG DB Link documentation.

The following items are described:

- *Abstract Data Type Descriptor*
- *Abstract Data Type Values*

Abstract Data Type Descriptor

Descriptor Class

A descriptor is used to describe abstract data types. This descriptor is implemented by the class `IldADTDescriptor`.

Such a descriptor can be embedded in an `IldDescriptor` instance if the described column or parameter is of a user-defined type, that is, when the function `IldDescriptor::getType` returns either `IldObjectType` or `IldCollectionType` objects.

Categories of User-Defined Data Types

A value of a user-defined data type is always an instance of `IldADTValue`, whether a “horizontal” structure (`object` for Oracle, `named row` or unnamed `row` for Informix Universal Server) or a “vertical” structure (`varray` or `nested table` for Oracle, `list`, `set`, or `multiset` for Informix US).

To differentiate between them, the user-defined data type descriptor can be queried using the function `getType`, which returns an `IldADTType` value, according to Table 4.4:

Table 4.4 *Categories of User-Defined Data Types*

This type	Is used for these data types
`IldADTObject`	Objects and named rows
`IldADTTable`	Nested tables
`IldADTList`	Lists, sets, and multisets
`IldADTArray`	Varrays

The type of a user-defined data type descriptor can only be a value from the enumeration `IldADTType` declared in the file `ild.h`. In other words, an `IldADTDescriptor` instance can represent one of the following:

- An Oracle `varray` if its type is `IldADTArray`,
- An Oracle `object` or an Informix US `named row` or unnamed `row` if its type is `IldADTObject`,
- An Oracle `nested table` if its type is `IldADTTable`, or
- An Informix US `list`, `set`, or `multiset` if its type is `IldADTList`.

Abstract Data Type Values

Values of an abstract data type are handled using instances of the class `IldADTValue`. Such a value keeps a reference to the abstract data type descriptor, which can be accessed by calling the function `IldADTValue::getDescriptor`.

Retrieving Values from the Result Set

An `IldADTValue` object is returned by the function `IldRequest::getColADTValue`. This object becomes the application's property. This means that the application must delete it once done with it.

Sending Values as Parameters

Once a parameter has been bound to the IBM ILOG DB Link types `IldObjectType` or `IldCollectionType` using the member function `IldRequest::bindParam`, values can be passed using the overloaded function `setParamValue`. The first parameter of this function can be deleted immediately after the call because IBM ILOG DB Link copies it.

Accessing Attribute Values

Whether the object represents the value of an object type or the value of a collection type, the individual attribute or slot values are retrieved using the same accessor functions of the form `get<type>Value()` where `<type>` can be one of:

- `String`
- `Integer`
- `Real`
- `Byte`
- `Money`
- `Date`
- `Numeric`
- `DateTime`
- `Bytes`
- `ADT`
- `Ref`

Warning: *The values returned for* `string` *and* `ADT` *types must be copied by the application.*

Extending the IldRequest Class

It is not possible to derive from the class `IldRequest` because the actual objects handled by your application are not instances of the `IldRequest` base class but instances of *subclasse*s of this class.

If, for any reason, you need to extend the functionality of the IBM ILOG DB Link class `IldRequest`, you can retrieve the determining part for handling statements.

The member function `IldRequest::getHook` returns it as a void pointer (`IlAny`). The actual return value depends on the target RDBMS, as shown in Table 4.5:

Table 4.5 *Values Returned by the Member Function IldRequest::getHook*

With this Database System...	...the `getHook` function returns	...and lets you call the following database proprietary client interface.
DB2	the SQLHANDLE	CLI (Call Level Interface)
Informix	the cursor name	Embedded SQL

Table 4.5 *Values Returned by the Member Function IldRequest::getHook*

With this Database System...	...the `getHook` function returns	...and lets you call the following database proprietary client interface.
MS SQL Server	the pointer to the `DBCURSOR` structure	DB Library function
ODBC	the `HSTMT`	ODBC functions
OLE-DB	the pointer to the `IDBCreateCommand` structure	OLE-DB functions
Oracle 9i, 10g and 11g	the pointer to the `OCIStmt` structure	OCI function
Sybase	the pointer to the `CS_COMMAND` structure	Client Library functions

Differences between IldRequest and IldRequestModel Classes

`IldRequestModel` provides the same functionalities as `IldRequest`, plus the ability to be derived. This derivation capability introduces a few differences with the `IldRequest` class, as follows:

- `IldRequestModel` instances are allocated using their constructor, and not using the `IldDbms::getFreeRequest` member function of the `IldDbmsModel` class.
- `IldRequestModel` objects are not automatically cleaned when deleting the `IldDbmsModel` instance through which they were allocated.

Queries

This chapter is divided into the following sections:

- *Executing an SQL Query Immediately*
- *Setting Up a Query for Multiple or Repeated Use*
- *Binding Application Memory to the Database API*
- *Finding Out the Types and Sizes of Returned Columns*
- *Retrieving Data*

Executing an SQL Query Immediately

Some SQL statements sent by your application can be:

- "one shot" queries, such as DDL statements (`create table`, `drop table`, and so on),
- or, `select` or `delete` queries, known beforehand.

For immediate execution, you do not even need to create an `IldRequest` object. The `IldDbms` object can process such statements by means of the function `IldDbms::execute`.

However, you can also use an `IldRequest` object for that purpose by calling the function `IldRequest::execute`.

When the execution succeeds, you may bind the returned data columns to application memory by using the member function `IldRequest::bindCol`.

Setting Up a Query for Multiple or Repeated Use

When your application reuses the same basic SQL statement several times, at once or in various places, with different values as parameters, you can allocate an `IldRequest` object, which will be used to prepare the query for execution and will be kept until actual execution is needed.

Such a query contains placeholders in the form of question marks, "`?`", as defined in the ISO SQL standard, or even in an RDBMS-specific form, such as "`:1`" or "`:var`". These last two forms are supported only by Oracle which does not support the standard placeholder format. As a consequence, queries that need to be prepared cannot be made portable on all RDBMSs and your application code must allow for this.

The general procedure to set up a query for multiple or repeated use is the following:

1. Call the function `IldRequest::parse` to prepare a parameterized query.

   Once the query has been prepared successfully, you must bind the variables (which are the program counterpart of the placeholders) to the proper type and, possibly, to the application memory.

2. To do so, use the member function `IldRequest::bindParam`.

   Before actual execution, set the variable values either in your application memory or in the IBM® ILOG® DB Link internal memory.

3. To do so, use the member function `IldRequest::setParamValue`.

   An overloaded version of that member function exists for each possible IBM ILOG DB Link type.

When there are many rows to be inserted, this schema can be implemented in two different ways:

- As a loop inserting one row at a time.

  For each row of parameter values, call `IldRequest::setParamValue` for all parameters, then call `IldRequest::execute`.

- As a batch inserting several rows at once.

4. Using the array bind mode, pass all parameter values, up to the number of rows indicated by `IldRequest::getParamArraySize`.
5. Call the member function `IldRequest::execute`.

   If there are fewer rows than the bind array size, you can pass the actual number of rows as the second parameter to the function `IldRequest::execute`.

Binding Application Memory to the Database API

On both input and output, you can bind application memory to the API of the RDBMS by calling the function `IldRequest::bindParam` for input or `IldRequest::bindCol` for output.

- Input bindings must be done after the query has been successfully prepared.
- Output bindings make sense only if the query is an SQL `select` statement or an SQL `execute procedure` statement that returns rows. In the case of immediate execution, output bindings can be done after the execution and before the first call to the member function `IldRequest::fetch`.

In the case of a prepared query, the bindings can take place after the function `IldRequest::execute` has been called.

Finding Out the Types and Sizes of Returned Columns

Once the `select` statement has been prepared using the function `IldRequest::parse` or executed using the function `IldRequest::execute`, the application can gain access to the column descriptors.

The number of `select list` columns is given by a call to the member function `IldRequest::getColCount`. Then, for any index between 0 and that value, the function `IldRequest::getColDescriptor` returns an instance of the class `IldDescriptor`.

Table 4.2 on page 74 shows the `IldDescriptor` information you can get on the corresponding column.

Retrieving Data

Depending on your implementation, the application can either retrieve data in its own memory space or let IBM® ILOG® DB Link handle all memory allocation and data retrieval.

The simplest way is to process the SQL statements and then ask IBM ILOG DB Link for the column data values via the function `IldRequest::getCol<data type>Value`, where `<data type>` is one of IBM ILOG DB Link-supported types. All accessors check their argument values. Therefore, the data type for which the accessor is made must match the actual column binding type, and the indexes to the result set, column, and row numbers (optional) must not be out of bounds.

When you use accessors to data types like `string` or `binary`, you must copy the returned value to application-allocated memory. This is done because the next data fetch reuses the same internal memory, and, consequently, the previously fetched data will be lost.

C H A P T E R

6

Errors and Warnings

This chapter explains the error handling mechanism implemented by IBM® ILOG® DB Link and covers the following topics:

- Diagnostic Class — The class `IldDiagnostic`.
- Warnings — Querying the classes `IldDbms` and `IldRequest` on various information messages.
- Errors — Error codes and messages as raised by IBM ILOG DB Link.
- Error Reporter — Default settings and behavior of the IBM ILOG DB Link error handling mechanism, and the output error stream.
- Customizing the Error Handling Mechanism

Diagnostic Class

The `IldDiagnostic` class contains information about the context of an error or warning. The following items are described:

- Accessing a Diagnostic Instance
- Context Information

Accessing a Diagnostic Instance

The information provided by a given instance of `IldDiagnostic` is relevant only if the function `IldIldBase::isErrorRaised` or `IldIldBase::isInformationRaised`:

- has been called before the function `IldIldBase::getError` or `IldIldBase::getInformation`, respectively, *and*
- returned `IlTrue`.

Otherwise, the `IldIldBase::getError` or `IldIldBase::getInformation` function may return either null or an instance of `IldDiagnostic` whose contents are irrelevant.

If any context information items need to be kept, the application must copy the corresponding values into its own memory space because the contents of the `IldDiagnostic` object might be overwritten during subsequent execution of a query.

Context Information

The object contains the following information items about the context in which it was created or filled:

- Code: This number can originate from IBM® ILOG® DB Link or from the server.
- Native code of the error: When the error is raised by IBM ILOG DB Link, it is set to 0. Otherwise, it has the same value as the code.
- Function code: The IBM ILOG DB Link symbolic code for the function in which the error was raised.
- Origin: The layer that raised the error.
- sqlstate: Either the value returned by the server, the value set by IBM ILOG DB Link, or no value at all.
- Message text: The text associated with the error.

Warnings

Objects of the classes `IldDbms` and `IldRequest` can be queried about the arrival of warnings or other information messages.

When such a message is received, a flag is raised in the object. To test the flag, call the function `IldIldBase::isInformationRaised`. If the return value is `IlTrue`, a warning or information message has been received. You can read its symbolic code and text by calling the member function `IldIldBase::getInformationCode` or `IldIldBase::getInformationMessage`. For example:

```
if (request->isInformationRaised()) {
   cout << "Information: " << endl;
   cout << "\t" << request->getInformationMessage() << endl;
}
```

Platforms: When your application is connected to Sybase, these information messages originate from the `TransactSQL` function `print`.

Errors

This section is divided as follows:

- *Error Handlers*
- *Error Codes*
- *IBM ILOG DB Link API Codes and Messages Table*
- *Function Codes*
- *SQLSTATE*
- *Error Messages*
- *Error Origin*
- *Erroneous IldDbms and IldRequest Objects*

Error Handlers

In the process of exchanging information with a database system, there are numerous opportunities for errors. These can be raised by IBM® ILOG® DB Link, by the client API of the RDBMS, or even by the database server.

IBM ILOG DB Link implements error handlers to catch errors. All `IldDbms` and `IldRequest` objects must have an attached `IldErrorReporter` object. In fact, when `IldDbms` and `IldRequest` objects are created, an error handler is automatically attached.

However, you can change the default error handler for an object of a class derived from `IldErrorReporter`.

> ***Note:*** *It is not possible to access the default reporter. Thus, when no reporter has been set by the application, although the function* `IldIldBase::getErrorReporter` *returns null, a reporter is actually set.*

Platforms: When using Sybase, your application must not replace the message or error handlers by the Sybase library routine `ct_callback` with the parameter `CS_CLIENTMSG_CB` or `CS_SERVERMSG_CB`. Such a call made by your application would break the error handling mechanism of IBM ILOG DB Link. The same problem occurs with MS SQL Server and the `DB Library` functions `dberrhandle` and `dbmsghandle`. Your application must conform to the protocol defined in *Customizing the Error Handling Mechanism* on page 116.

Error Codes

ILOG DB Link defines error codes associated with error messages. Some of these codes correspond to anomalies detected by the library. Additionally, there are error codes and messages corresponding to those raised within an RDBMS itself. ILOG DB Link provides the necessary interface for your application to recover gracefully (whenever possible) from these two kinds of errors in the class `IldErrorReporter`.

Error codes returned by the member function `IldIldBase::getErrorCode` may originate from IBM ILOG DB Link or from the RDBMS. You cannot tell the origin of the error from that number only, because IBM ILOG DB Link error codes are negative, just like most RDBMS error codes.

However, you can differentiate between them through the error origin: the function `IldIldBase::getErrorOrigin` returns `IldDblink` if the error was trapped by the IBM ILOG DB Link API, or `IldRDBMServer` if the error was trapped by the server. Moreover, Sybase allows you to distinguish the errors raised in the server from those raised in the client API: for the latter, the origin is of type `IldClientAPI`.

IBM ILOG DB Link API Codes and Messages Table

The following table gives you a list of all error messages generated by the IBM ILOG DB Link API together with the corresponding error codes. The reasons for the errors are also included.

Table 6.1 *Error Codes and Messages*

Error Code (Symbol)	Error Message (String)	Comment
`ILD_ALREADY_CONNECTED`	*Current object is already connected*	This error can be raised by the function `IldDbms::connect` if an active connection is already open.
`ILD_BAD_COLUMN_INDEX`	*Bad column index*	This error is raised by the function `IldRequest::getColName` when the column descriptor has no name or the given index is out of bounds.
`ILD_BAD_COLUMN_NAME`	*Bad column name*	This error is raised by the function `IldRequest::getColIndex` when no column with the given name is found in the results set descriptors.
`ILD_BAD_DB_SPEC`	*Bad format for database specification*	This error is raised at connection time when the given connection string does not match the RDBMS-specific format requested.
`ILD_BAD_EXECUTE_COUNT`	*Bad count for execute function*	This error is raised when the function `IldRequest::execute` is called for a prepared query whose second argument is greater than the size of the bind array.
`ILD_BAD_FILE`	*File cannot be opened for write operation*	This error is raised by the function `IldRequest::getLargeObject` when the file indicated by the *fileName* argument is write-protected.
`ILD_BAD_VARIABLE_SIZE`	*Bad size for variable being bound*	This error is raised when binding a column with a byte size that is too small for the data type used.
`ILD_CANNOT_RESIZE_TUPLE`	*Cannot resize tuple*	This error is raised while describing a results set or binding a parameter when an internal allocation failed
`ILD_CBCK_INIT`	*Callback initialization failed*	This error can be raised only within a connection to Sybase. If such an error occurs, contact IBM customer support.

Table 6.1 *Error Codes and Messages (Continued)*

Error Code (Symbol)	Error Message (String)	Comment
ILD_CON_ALLOC	*Connection allocation failed*	This error can be raised only within a connection to Sybase. If such an error occurs, contact IBM customer support.
ILD_CON_INIT	*Connection initialization failed*	This error can be raised only within a connection to Sybase. If such an error occurs, contact IBM customer support.
ILD_CTXT_ALLOC	*Context allocation failed*	This error can be raised only within a connection to Sybase. If such an error occurs, contact IBM customer support.
ILD_CTXT_INIT	*Context initialization failed*	This error can be raised only within a connection to Sybase. If such an error occurs, contact IBM customer support.
ILD_DATE_CONVERT	*Date conversion failed*	This error is raised when a conversion fails, either from the RDBMS internal format to a date value or, conversely, from a date value to the RDBMS internal format.
ILD_DBMS_FATAL_ERROR	*Fatal Dbms error*	This error is raised after an unrecoverable error occurs. After the error is raised, the connection is in an unpredictable state and must not be reused. The `IldDbms` object must be destroyed.
ILD_DBMS_NOT_CONNECTED	*Dbms is not connected*	This error is raised each time a function from the class `IldDbms` is called while the connection is closed.
ILD_IGN_EXT_ROWS	*Extra row(s) ignored*	WARNING ONLY This is a warning emitted by member functions such as `IldRequest::getLargeObject` when the given condition is not restrictive enough and several rows are returned.
ILD_INVALID_HANDLE	*Invalid handle*	This error is raised when the underlying control structure used to communicate with the server is out of order.

Table 6.1 *Error Codes and Messages (Continued)*

Error Code (Symbol)	Error Message (String)	Comment
ILD_INVALID_PARAMETER	*Data exception, invalid parameter value*	This error is raised when a function of IBM ILOG DB Link is called with an invalid value. For example, the *userCallBack* parameter (which is the user function called when the event occurs) in the `IldDbms::subscribeEvent` function can not be `null`. If it is, the program will crash when the event occurs.
ILD_INVALID_SEQUENCE	*Calling this function is not allowed at this time*	This error is raised when a function is called when some other function should have been called prior to this one.
ILD_LIB_MSMTCH	*Library mismatch*	If your application was linked in dynamic mode, this error is raised when the driver manager finds a library with the right name but with no proper entry point.
ILD_LIB_NLNKD	*RDBMS library not linked*	If your application was linked in static mode, this error is raised when the target RDBMS library has not been linked or the RDBMS name is not recognized.
ILD_LOCK_NAME_MISMATCH	*Lock name mismatches*	This error can be raised only when an attempt is made to unlock an `IldRequest` object using the wrong name or an empty name for the lock to be released.
ILD_MAX_CURS_LEN	*Cursor name truncated*	WARNING ONLY This warning is raised when the name passed for a cursor is too long for the target RDBMS.
ILD_MEMORY_EXHAUSTED	*Memory exhausted*	This error is raised when an allocation fails.
ILD_NO_DYN_LIB	*Dynamic library not found*	If your application was linked in dynamic mode, this error is raised when the driver manager cannot find the designated driver library.
ILD_NO_HANDLER	*Error handler not called*	This error can be raised only within a connection to Sybase. If such an error occurs, contact IBM customer support.
ILD_NO_MORE_TUPLES	*No more tuples*	This error is raised when an accessor to column data is used and no row has been returned from the server.

Table 6.1 *Error Codes and Messages (Continued)*

Error Code (Symbol)	Error Message (String)	Comment
ILD_NO_REPORTER	*Error reporter cannot be null*	This error is raised by the functions `IldDbms::setErrorReporter` and `IldRequest::setErrorReporter` when the argument passed is `null`.
ILD_NOT_IMPLEMENTED	*Not implemented for current RDBMS*	This error is raised when an attempt is made to use a functionality that cannot be implemented for the target RDBMS.
ILD_NOT_SCROLL_MODE	*Scrollable cursor mode must be activated*	This error is raised when an attempt is made to use the function `IldRequest::fetchScroll` in the following context: A value of the *fetchOrientation* parameter is different from `IldFetchDirectionNext`. Scrollable cursor mode is not activated (see member function `IldRequest::setScrollable`).
ILD_NUM_CONVERT	*Numeric conversion failed*	This error is raised when a conversion fails, either from the RDBMS internal format to a numeric value or, conversely, from a numeric value to the RDBMS internal format.
ILD_OFFSET	*Offset*	INFORMATION ONLY This is not an error but merely part of an error message.
ILD_OUT_OF_RANGE	*Index out of range*	This error is raised when an attempt is made to bind an output column using an index greater than the actual number of columns in the results set.
ILD_RDBMS_CONN	*Must give RDBMS name and connection string*	This error is raised when trying to create an `IldDbms` object and either the RDBMS name or the connection string was `null` or started with a `null` character.
ILD_REQUEST_REQUIRED	*`IldRequest` object missing*	This error can be raised only when connected to Sybase, or MS SQL Server. With these RDBMSs, the transaction-handling `IldDbms` functions require that a valid `IldRequest` object be passed.

Table 6.1 *Error Codes and Messages (Continued)*

Error Code (Symbol)	Error Message (String)	Comment
ILD_TYPE_MISMATCH	*Value accessor mismatch*	This error is raised if an output or input column is bound to a type, in the IBM ILOG DB Link sense, that is not possible or not allowed by the configuration settings (for example, binding a column to `IldDateType` while the `IldRequest` object is set to use the “date as object” feature).
ILD_UNCHGEABLE	*Cannot modify a server initialization parameter*	This error can be raised only within a connection to Oracle, when an attempt is made to modify the request time-out.
ILD_UNDEF_LINK_MODE	*Unknown DB Link driver linkage mode*	The application must be linked with either the `dblnkdyn` or `dblnkot` library. If none is used, this error is raised when trying to allocate a connection.
ILD_UNKN_ERRMSG	*Unknown error message*	REPLACEMENT MESSAGE This is a replacement message, not an error. It is raised when the RDBMS API fails to give the proper message for an error.
ILD_UNKNOWN_CODE	*Unknown error code*	This error is raised when a function call to the underlying RDBMS API returns an unexpected value. If such an error occurs, contact IBM customer support.
ILD_UNKNOWN_ENTITY	*Unknown relation*	This error is raised when a schema entity description is not available because it does not exist in the database.
ILD_UNKNOWN_RDBMS	*Unknown RDBMS*	This error is raised when an `IldDbms` object is created and the given RDBMS name is not recognized by the driver manager. It can be raised only if the application is linked in dynamic mode.
ILD_UNKNOWN_TYPE	*Unknown column type*	This error is raised if an attempt is made to bind an output column or an input parameter to a type that is not supported for the target RDBMS.

Table 6.1 *Error Codes and Messages (Continued)*

Error Code (Symbol)	Error Message (String)	Comment
ILD_USING_ERROR_DBMS	*Using Error Dbms object*	This error is raised by any function of the class IldDbms when this function is used against an object that was not properly built. This occurs when an error is raised because an error occurred when the IldDbms object is created or when the initial connection is processed.
ILD_USING_ERROR_REQUEST	*Using Error Request object*	This error is raised by any function of the class IldRequest when the object has not been allocated properly, usually because the attached IldDbms object has not been connected to the server.

Function Codes

Each documented function has a unique identifier whose symbolic name mimics its name. For example, the identifier for the function `IldDbms::disconnect` is `ILD_D_DISCONNECT`. These identifiers are defined by the enumeration type `IldFuncId` in the file `ild.h`, under the section "`Db Link Function Ids`."

SQLSTATE

Whenever the RDBMS gives access to the `SQLSTATE` value, IBM ILOG DB Link registers that value. The text of the `SQLSTATE` message can be retrieved using the member function `IldIldBase::getErrorSqlstate`. The content of the text is volatile —that is, it may be overwritten by other data. Therefore, you must copy it if you want to keep a trace of the error.

Error Messages

The text of an error message can be retrieved using the member function `IldIldBase::getErrorMessage`. The content of the text is volatile, that is, it may be overwritten by other data. Therefore, you must copy it if you want to keep a trace of the error.

For error message strings, see Table 6.1 on page 107.

Error Origin

The enumeration type `IldErrorOrigin` indicates the various sources of errors: IBM ILOG DB Link itself, the client API of the RDBMS, or the database server.

Depending on the layer in which the error was raised, IBM ILOG DB Link sets the error origin to a different value of `IldErrorOrigin`.

```
enum IldErrorOrigin {
     IldUnknownOrigin,
     IldDbLink,
     IldClientAPI,
     IldRDBMServer
};
```

When the error is raised by an IBM ILOG DB Link function, the origin is set to `IldDbLink`.

Platforms: When connected to a Sybase server, the origin will be set to `IldClientAPI` if the `CLIENT` callback has been activated. Otherwise, it is set to `IldRDBMServer`.

Erroneous IldDbms and IldRequest Objects

If a very serious error is raised when an object is created, the return value you receive may be of the class `IldErrorDbms` or `IldErrorRequest` (instead of the object you expect).

Instances of these classes will never process any SQL statement. They always raise an error, either `ILD_USING_ERROR_DBMS` or `ILD_USING_ERROR_REQUEST`.

- `ILD_USING_ERROR_DBMS`: An erroneous connection object is returned only when an `IldDbms` object cannot be allocated.
- `ILD_USING_ERROR_REQUEST`: An erroneous request object is returned only when you try to get a new `IldRequest` object even though the related `IldDbms` object is not connected.

Warning: *You must explicitly delete these erroneous objects.*

Error Reporter

This section explains how an error reporter is created and how an output error stream can be attached to it. It is divided as follows:

- Default Settings and Behavior
- Output Error Stream

Default Settings and Behavior

When an `IldDbms` object is created, an `IldErrorReporter` object is attached to it. This object is inherited by all `IldRequest` objects created from that `IldDbms` object.

The output stream field of the `IldErrorReporter` object is not set on creation.

IBM ILOG DB Link guarantees that, when an error is raised, the `IldErrorReporter` object will call one of the following member functions:

- `IldErrorReporter::dblinkError` if the error was raised by the IBM ILOG DB Link API, or
- `IldErrorReporter::dbmsError` if the error was raised by the database server or the RDBMS client API.

Output Error Stream

An output stream (an instance of the C++ class `ostream`) can be attached to an `IldErrorReporter` object. Once an output stream has been attached, the error handling mechanism will output the code and text of the error through that stream.

An output stream is attached by a call to the function `IldErrorReporter::setOStream` and can be retrieved by a call to the function `IldErrorReporter::getOStream`.

The error code and text are formatted in the output stream like this:

```
<function name> = <error code> = <error text>
```

where `<function name>` indicates the IBM ILOG DB Link member function (from the list in Table 6.2) in which the error was raised:

Table 6.2 *Error-raising Member Functions in the Class IldDbms*

Class IldDbms		
`Constructor`	`autoCommitOn`	`autoCommitOff`
`commit`	`connect`	`disconnect`

Table 6.2 *Error-raising Member Functions in the Class IldDbms*

Class IldDbms		
execute	getAbstractType	getHook
getInfo	getName	getProcedure
getRelation	getSynonym	getTypeInfo
readAbstractTypeNames	readEntityNames	readOwners
readRelationNames	readRelationOwners	readProcedureNames
readSynonymNames	readPrimaryKey	readForeignKeys
readIndexes	readSpecialColumns	rollBack
setCursorMode	setErrorReporter	setTimeOut
startTransaction		

Table 6.3 *Error-raising Member Functions in the Class IldRequest*

Class IldRequest		
Constructor	Destructor	bindCol
bindParam	closeCursor	execute
fetch	getColADTValue	getColBinaryValue
getColByteValue	getColDateTimeValue	getColIndex
getColIntegerValue	getColLongTextValue	getColMoneyValue
getColName	getColNumericValue	getColRealValue
getColReferenceValue	getLargeObject	getParamBinaryValue
getParamCursorValue	getParamDateTimeValue	getParamIndex
getParamNumericValue	getStatus	insertLongText
insertBinary	isColNull	isNullIndicatorOn
isParamNull	parse	release
removeColLock	removeColArraySize	removeParamArraySize
removeParamLock	setColArraySize	setColPos

Table 6.3 *Error-raising Member Functions in the Class IldRequest*

Class IldRequest		
setCursorName	setErrorReporter	setParamArraySize
setParamNullInd	setParamValue	setReadOnly

Customizing the Error Handling Mechanism

IBM ILOG DB Link assumes that the client application will have to run 24 hours a day, seven days a week, so it provides a full error-handling mechanism that does not allow the application to exit prematurely. This mechanism captures warning messages and errors. Its mainspring is an object of the class `IldErrorReporter`.

To a certain extent, you can customize the mechanism by subtyping this class. Doing so enables you to set your own error reporters on every object of the classes `IldDbms` and `IldRequest`, by using the member functions `IldDbms::setErrorReporter` or `IldRequest::setErrorReporter`. It is an error to try to set a null value as the error reporter since the error `ILD_NO_REPORTER` is then raised in the calling object.

Customizing the error handling mechanism is done in two main steps:

1. Derive from the class `IldErrorReporter` and define the virtual functions `IldErrorReporter::dbmsError` and `IldErrorReporter::dblinkError`:

```
class UserErrorReporter: public IldErrorReporter {
public:
virtual void dbmsError(IlInt,
                       const char*,
                       const char*,
                       IldDbms*,
                       IldRequest* = 0,
                       const char* = 0) const;
virtual void dblinkError(IlInt,
                         const char*,
                         const char*,
                         IldDbms*,
                         IldRequest* = 0,
                         const char* = 0,
                         IlInt = (IlInt)0,
                         const IldRelation* = 0) const;
};
```

2. Create an instance of your derived class and attach it to the IBM ILOG DB Link object you choose.

```
// The following UserErrorReporter objects will be
// destroyed by the IldDbms and IldRequest destructors
// respectively.
UserErrorReporter* dbmsReporter = new UserErrorReporter;
```

```
UserErrorReporter* requestReporter = new UserErrorReporter;
...
cout << "Error reporter set to user defined one" << endl;
dbms->setErrorReporter(dbmsReporter);
...
cout << "Setting request error reporter to user defined one" << endl;
request->setErrorReporter(requestReporter);
```

Base Class

In objects of the class `IldErrorReporter`, the following fields can be set when an error is raised:

```
IldDbms*          _dbms;
IldRequest*       _request;
IldRelation*      _relation;
const char*       _query;
IlInt             _index;
IlInt             _size;
```

An accessor function exists for each of these fields. For example, the field `_dbms` is read with the accessor `IldErrorReporter::getDbms`. The same naming convention is used for all fields.

Most fields also have a modifier function, but it is neither advisable nor practical to use it. After the whole internal processing of the error is complete, one of the two member functions `IldErrorReporter::dblinkError` or `IldErrorReporter::dbmsError` is called.

Virtual Functions and Their Parameters

- The member function `IldErrorReporter::dbmsError` takes the following arguments:
  - the function code,
  - the function name,
  - the error text,
  - the `IldDbms` object (if appropriate),
  - the `IldRequest` object (if appropriate),
  - the connection string (if appropriate).

```
void UserErrorReporter::dbmsError(IlInt errorType,
                                  const char* function,
                                  const char* message,
                                  IldDbms* dbms,
                                  IldRequest* request,
                                  const char* string) const;
```

- The member function `IldErrorReporter::dblinkError` takes the following arguments:
  - the function code,
  - the function name,
  - the error text,
  - the `IldDbms` object (if appropriate),
  - the `IldRequest` object (if appropriate),
  - the connection string (if appropriate),
  - the index value (if appropriate),
  - the `IldRelation` object (if appropriate).

```
void UserErrorReporter::dblinkError(IlInt errorType,
                                    const char* function,
                                    const char* message,
                                    IldDbms* dbms,
                                    IldRequest* request,
                                    const char* string,
                                    IlInt index,
                                             const IldRelation* relation)
                                    const;
```

In both lists, “if appropriate” means that these arguments can have a null value if the function where the error is raised does not use such an object. One of the two arguments `dbms` and `request` must be non-null.

C H A P T E R

7

Compiling and Linking

This chapter deals with compilation and compatibility issues, and presents the IBM® ILOG® DB Link libraries. It is divided as follows:

- *Compilation Flags*
- *Target RDBMSs*
- *IBM ILOG DB Link Libraries*

> ***Warning:*** *The* IBM ILOG DB Link *include file "*`dblink.h`*" must always be included in your application. It defines a static variable that will initialize the driver linkage mode (either static if you use* `dblnkst` *+ RDBMS compilation flag, or dynamic if you use* `dblnkdyn`*).*

Compilation Flags

IBM ILOG DB Link can be present in an application using two different protocols:

- The target RDBMSs are known at compile-link time:

  RDBMS-specific flags are used at compile time and the IBM® ILOG® DB Link drivers corresponding to the target RDBMSs are linked to the application, together with the RDBMS client libraries.

- The application is generic with respect to the RDBMSs:

  No compile-time flag is used. The link-time options only refer to the IBM ILOG DB Link driver manager library.

> ***Warning:*** *Never mix both protocols in the same application, because linking with the driver manager is incompatible with "static" linkage of the drivers.*

When you use IBM ILOG DB Link to write an application dedicated to a specific RDBMS, you must set some specific compiler flags. These flags depend on the target RDBMS and on the mode in which the code will be linked.

While reading the header files, IBM ILOG DB Link defines other flags that you can use to achieve portability.

This section is divided as follows:

- *Compatibility with Previous Releases*
- *RDBMS Flags*
- *Dynamic Load*
- *Mode and Flag*

Compatibility with Previous Releases

Some changes in IBM ILOG DB Link V5.3 cause incompatibilities with code written for IBM ILOG DB Link 4.x. The current version is not binary compatible with previous ones.

Generic Data Types

IBM ILOG DB Link V5.3 no longer defines its own basic data types (`IldInt`, `IldShort`, and so on). Instead, it uses the IFC (IBM ILOG Foundation Classes) data types (`IlInt`, `IlShort`, and so on).

If you want to port an application that was developed with IBM ILOG DB Link 4.x to IBM ILOG DB Link V5.3, you need to make the changes listed in the following table.

Here is the list of changes to implement for the conversion:

Table 7.1 *New Macros for IBM ILOG DB Link V5.3*

Old Macro	New Macro
ILDWINDOWS	WINDOWS
ILDSTD	IL_STD
ILDSTDUSE	IL_STDUSE
ILDSTDPREF	IL_STDPREF
ILD_MAX_NUM_LEN	IL_MAX_NUM_LEN

Table 7.2 *IFC Generic Data Types for IBM ILOG DB Link V5.3*

Old Data Type	New Data Type
IldBoolean	IlBoolean
IldFalse	IlFalse
IldTrue	IlTrue
IldInt	IlInt
IldUInt	IlUInt
IldAny	IlAny
IldUShort	IlUShort
IldByte	IlUChar
IldNumeric	IlNumeric

Library Organization

The organization of libraries has also changed.

- With previous releases, there was one library for each supported database plus the library `dblink` to support dynamic loading.
- This release contains:
  - A specific library that contains the IBM ILOG DB Link kernel: `dbkernel`.
  - Plus one library for each supported RDBMS.
  - Plus two libraries defining whether RDBMS libraries are linked statically or dynamically: `dblnkst` and `dblnkdyn`.

So if you want your application to be linked statically with, for instance, IBM DB2 and Oracle 9 drivers, it must be linked with:

```
dblnkst + dbdb2 + dbora9 + dbkernel
```

If you want the application to use dynamically loadable drivers, it must be linked with:

```
dblnkdyn + dbkernel
```

Important: *With certain systems, the order of the libraries is important. You must use first the library that defines the link mode (static or dynamic), then, if used, the IBM ILOG DB Link RDBMS driver, then finally the IBM ILOG DB Link kernel.*

RDBMS Flags

There is one compiler flag for each supported RDBMS:

- DB2: `ILDDB2`
- Informix: `ILDINFORMIX`
- MS SQL Server: `ILDMSSQL`
- ODBC: `ILDODBC`
- OLE-DB: `ILDOLEDB`
- Oracle: `ILDORACLE`
- Sybase: `ILDSYBASE`

There must be at least one of these flags per compilation to allow the corresponding driver to be effectively linked at link time. If not, no error will be issued at compile time or at link time. At runtime, however, it will not be possible to create any connection.

The makefiles for all examples have a variable that defines one of these flags:

```
DBMSCCFLAGS=-DILDORACLE
```

Dynamic Load

IBM ILOG DB Link is delivered as a set of libraries that include:

- a kernel library: `libdbkernel.a` (or `dbkernel.lib`)
- a dynamic driver manager: `libdblnkdyn.a` (or `dblnkdyn.lib`)
- a static driver manager: `libdblnkst.a` (or `dblnkst.lib`)
- a set of drivers, possibly several per support.

On PCs, there is no difference between a driver and the delivered DLL.

The kernel and dynamic load libraries (`[lib]dbkernel` and `[lib]dblnkdyn`) are themselves dynamically loadable on all UNIX® platforms.

On UNIX, the delivered shared libraries are **not** built as dynamically loadable drivers. On a per-platform basis, a makefile named `Makefile.drv` is delivered, which allows you to build the drivers using the delivered object files. You cannot build a driver unless you have installed the RDBMS client libraries in a version that includes shared libraries. Thus, if the target RDBMS is Informix, you need at least version 7.2, and if the target is Oracle you need at least version 9.0. See *Building Dynamically-Loadable Drivers under UNIX* on page 126 for more information.

> ***Warning:*** *The only variables that can be modified in these makefiles are those regarding the list of RDBMS-dedicated libraries. The object files list must* **not** *be modified nor must the driver manager library be added to the library list.*

The drivers can be loaded only if their access path is present in an environment variable `LD_LIBRARY_PATH`.

Use of the dynamic load facility is demonstrated by a number of example files, which you can find in the directories `examples/dblink/<port name>`. The makefiles in these directories build the same examples as the ones that can be found in the dedicated directories. The makefiles in the `dblink` database differ from the others in that the `DBMSCCFLAGS` and `DBMSLDFLAGS` variable are empty, and the `DBLIB` variable is set to `dblink`.

Mode and Flag

IBM ILOG DB Link is delivered as a set of libraries compiled in various compilation modes.

- For UNIX®, there are two compilation modes per port: `static_pic`, and `shared` (and some variants like `static_stl` and `shared_stl` under AIX, and so on.) When your application is linked with the shared mode library, be sure you add the path to the library in the environment variable `LD_LIBRARY_PATH` before running it.
- For Windows® (NT, 2000, XP, or Vista), there are at least three compilation modes: `stat_mta`, `stat_mda`, `dll_mda`, with the compiler flag `IL_STD`. At runtime, the `PATH` environment variable must indicate the directory where the IBM ILOG DB Link library is installed.

Target RDBMSs

This section deals with:

- *Multiple Targets*, and
- *RDBMS Prerequisites*.

Multiple Targets

When the application is targeted for several database systems, just add the proper compilation flags and the proper RDBMS client libraries.

RDBMS Prerequisites

To build an executable application using IBM ILOG DB Link, you need to have the RDBMS client kit (available separately).

IBM ILOG DB Link Libraries

This section provides a table of the IBM® ILOG® DB Link libraries and draws your attention to the special `make` files supplied to enable you to rebuild dynamically-loadable drivers. It is divided as follows:

- *Library Names*
- *Building Dynamically-Loadable Drivers under UNIX*

Library Names

Table 7.3 provides the names of IBM ILOG DB Link libraries. These names have been stripped of their suffix. The suffix depends on the operating system and compilation mode. It can be one of:

- `.a`,
- `.so`,
- `.sl`,
- `.lib`, or
- `.dll`.

Table 7.3 IBM ILOG DB Link *Libraries*

RDBMS	UNIX® Name	Windows® Name
IBM ILOG DB Link driver manager	libdbkernel	dbkernel
Static Load Mode	libdblnkst	dblnkst
Dynamic load mode	libdblnkdyn	dblnkdyn
DB2	libdbdb2	dbdb2
DB2	libdbdb29x	dbdb29x
Informix Universal Server	libdbinf9	dbinf9
Microsoft® SQL Server®	-	dbmssql
ODBC	-	dbodbc
OLE-DB (for Microsoft SQL Server)	-	dboledb
Oracle v9i	libdbora9	dbora9
Oracle v10g	libdbora10	dbora10
Oracle v11	libdbora11	dbora11
Sybase	libctsyb	ctsyb

Building Dynamically-Loadable Drivers under UNIX

The delivery includes special makefiles to rebuild the dynamically-loadable drivers. These files are named `Makefile.drv`.

For all ports, these files use variables to locate the RDBMS client libraries. These variables follow the UNIX® convention of each RDBMS:

- `DB2DIR` for DB2
- `INFORMIXDIR` for Informix
- `ORACLE_HOME` for Oracle
- `SYBASE` for Sybase

The delivery includes the object files needed to rebuild the drivers. These files are separated into two groups: the IBM ILOG DB Link kernel files and the RDBMS-specific files. The IBM ILOG DB Link kernel files are always the same but the RDBMS files change depending on the client libraries.

On UNIX ports, you need to set the value of RDBMS dedicated variables:

- `INFLIBS` for Informix
- `ORALIBS` for Oracle
- `SYBLIBS` for Sybase

For the AIX port, the driver build process uses a special script called `makeC++SharedLib`, which is part of the compiler distribution.

To build the drivers, the client libraries must be "sharable" libraries—that is, `.so` files on UNIX®. The only exception is AIX, under which drivers can be built even from `.a` library files. If your current version of the client software does not include this type of library, building the drivers is impossible.

> ***Warning:*** *Do not mix RDBMS-specific files for one version with client libraries for another; even though the drivers building process may succeed, the runtime behavior is unpredictable and usually results in a memory fault.*

C H A P T E R

8

Code Samples

The IBM ILOG DB Link distribution includes a number of code samples that are delivered on an “as is” basis and are intended for information only. You can reuse their source code to implement parts of your application.

This chapter is divided as follows:

- *Generic Examples* — The examples presented in this section do not depend on your target RDBMs.
- *RDBMS-Specific Examples* — This section details a few sample files that were designed for the Informix, Oracle, and Sybase RDBMS respectively.

Generic Examples

The sample files presented in this section focus on the following IBM ILOG DB Link functionalities:

- *Basic Use* — An illustration of the simplest use that can be made of IBM ILOG DB Link libraries.
- *Handling Dates and Numbers*
- *SQL Interpreter* — A small-scale interpreter that sends (to the database server) the queries that the user types, and then retrieves the result set.
- *Concurrent Connections and Cursors*
- *Relation Searching* — Using the member function `IldDbms::getRelation`.
- *Relation Names* — Using the member functions `IldDbms::readRelationNames` and `IldDbms::readRelationOwners`.
- *Input Bindings* — Illustrating several combinations of input bindings.
- *Output Bindings* — How to use user-allocated and internally-allocated memory.
- *Multiple Output Bindings*
- *Handling LOBs* — How to use some specific `IldRequest` member functions.
- *Asynchronous Processing*

Basic Use

The `sample` example illustrates the simplest use that can be made of IBM ILOG DB Link libraries. It is independent of any RDBMS and is built from the files `sample.cpp` and `ildutil.cpp`.

It connects to the database server and checks whether the connection is properly established, and then creates an `IldRequest` object used to execute all SQL statements.

The SQL statements consist of:

- Creating a table: `CREATE TABLE`
- Inserting rows: `INSERT INTO`
- Selecting the whole table contents: `SELECT *`
- Finally, dropping the table: `DROP TABLE`

Neither the `insert` nor the `select` statements use parameters. They are executed immediately using the function `IldRequest::execute`.

After each call to this function, the error status is checked using the operator "`!`", which is applied to the reference to the caller returned by the function.

The disconnection and deletion of the `IldRequest` object are implicit: the deletion of the `IldDbms` object takes care of both aspects.

Handling Dates and Numbers

The `datasmpl` example shows how to handle dates and exact numeric values. It is built from the files `datasmpl.cpp` and `ildutil.cpp`. This example is RDBMS-dependent in that it changes the column data types according to the RDBMS name. See the global functions `IldGetDateTypeName` and `IldGetNumericTypeName` in the file `datasmpl.cpp`.

This example creates a table with two columns, the first holding numeric values, the second holding dates. For insertions, it uses the protocol for repeated execution with bound variables.

- The insertions are made with the *date as string* feature turned off. The second time the `IldRequest::bindParam` function is called, the value `IldDateTime` is passed as the column type argument.
- For the two first insertions, the exact numeric input is bound using the `IldStringType` column type. This choice allows you to send such values as "9876543210987.654321098" to the database server with no precision loss.
- For the following insertions, the *numeric as objects* feature is turned on. The second parameter is then bound using the `IldNumericType` value for the column type argument.
- After insertion, the table is read by three successive `select` statements.
  - For the first selection, the values are retrieved with the *numeric as string* feature turned on, and the date type values are retrieved using `IldDateTime` objects.
  - For the second selection, the *date as string* feature is turned back on.
  - For the third selection, the feature *numeric as objects* is turned back on.
- Before dropping the table, the previous cursor is explicitly closed using the function `IldRequest::closeCursor`. The `drop table` statement is executed using the `IldDbms::execute` function.

SQL Interpreter

The `ildsql` example is a simplified SQL interpreter that sends to the database server the queries that the user types, and then retrieves the result set (if any). It is built from the files `ildsql.cpp` and `ildutil.cpp`. This example is fully RDBMS-independent.

When retrieving a row from a result set, the example first checks that the column is not null for the current row.

Then, it dynamically calls the appropriate data accessors (`IldRequest::getCol<data type>Value` functions) as determined through the IBM ILOG DB Link type contained in the column descriptor (`IldRelation::getColType`). The code for the `IldPrintTuple` function is to be found in the file `ildutil.cpp`. This function makes use of all column type accessors.

This example can also handle more sophisticated statements like `commit` or `rollback`, and it even implements a table description facility through the command `describe <table name>`.

The kernel of the interpreter contains 35 lines of code, including error checking of each call to the RDBMS.

This example also supports the retrieval of multiple result sets because the fetch loop is doubled, that is, a first `do-while` loop retrieves the first row of a result set while the inner `while` loop fetches all remaining rows. When the inner loop stops due to a negative result from the call to `IldRequest::hasTuple`, the outer loop adds one more call to `IldRequest::fetch`, which starts retrieving the next result set, if any, and getting the result set column descriptions. If this call fails, no error is raised but the outer loop stops as well.

These nested loops are necessary because MS SQL Server, ODBC, and Sybase each have the capability to return several result sets for one `execute` call, which is the case when the SQL statement is a stored procedure call.

See the code in file `ildsql.cpp` for the other available options.

Concurrent Connections and Cursors

The example `multidb` illustrates the concurrence of connections and cursors. This example is built from the files `multidb.cpp` and `ildutil.cpp`.

- Three different connections are created as three `IldDbms` objects. Each of them has three cursors attached in the form of three `IldRequest` objects that are used to create tables and issue SQL `select` statements on these tables.
- Insertions are made into the tables using different cursors from the same connection.
- The tables are then fetched using the different cursors from a same connection for each table, the calls to `IldRequest::fetch` being intertwined.
- When the connection objects have been deleted, the cursor array is cleaned up. This is done to avoid keeping references to cursors that have become invalid because they have been deleted as a result of the corresponding connections being destroyed.
- The tables are then dropped using a new connection.

Relation Searching

The `readrel` example illustrates the use of the member function `IldDbms::getRelation` to retrieve a table description from the database schema.

The example is built from the files `readrel.cpp` and `ildutil.cpp`.

After connecting to the database server, this code sample tries to retrieve the description of a table that should not exist in the database.

Then, a table with a primary key is created before its description is retrieved and printed.

The table description displays the owner, name, and type of the relation. Its columns show the column name, native SQL type, size (in bytes), and nullability. Finally, the primary key and index descriptions are displayed.

The `IldPrintRelation` global function tries to get all possible keys and indexes from the table. It successively calls the `IldRelation` member functions `IldRelation::getPrimaryKey`, `IldRelation::getForeignKeys`, `IldRelation::getIndexes`, and `IldRelation::getSpecialColumns`.

Refer to the file `ildutil.cpp` to see the code for this function.

In the next step, the descriptor is deleted. Then, the `IldDbms` object is requested to get this descriptor using its index in the cache, hence an error.

An error is generated once more in the next step, which consists of dropping the table, and then trying to retrieve its description.

Relation Names

The `relnames` example illustrates the use of two `IldDbms` member functions:

- `IldDbms::readRelationNames`, both with and without the owner array output argument
- `IldDbms::readRelationOwners`

It is built from the files `relnames.cpp` and `ildutil.cpp`.

1. This sample file first queries the database server for all table names that get printed.
2. Then, the database is searched for all relation owner names.
3. Then, it retrieves all relation names and all their respective owner names.
4. Finally, it asks the user for an owner name that is used to query the database for all the names of all the relations belonging to that owner.

Each function returns one or two arrays of strings that are deleted using the function `IldDbms::freeNames`. Using this function is mandatory when running on a PC that uses

IBM ILOG DB Link libraries in DLL forms. Otherwise, a memory access error occurs during their deletion.

Input Bindings

The `smplbnd` example illustrates most of the possible combinations of input bindings. It is built from the files `smplbnd.cpp` and `ildutil.cpp`.

1. In the first step, rows are inserted, one by one, into a newly created table. This is achieved using immediate execution through the function `IldRequest::execute`.
2. Then, a `select` statement with a variable in the `where` clause is prepared for repeated execution. It is executed twice with different values for the variable.
3. Then, an `insert` statement, where all inputs are supplied through variables, is prepared for multiple execution. The *array bind* feature is set so that two rows will be inserted at once for each execution.

   One variable per row is set to null by the function `IldRequest::setParamNullInd`. After that insertion, a `select` statement is issued to check that the rows were inserted and the null values used despite the values that were actually passed to the parameter.
4. Finally, an `update` statement with parameters is prepared and executed, but in the call to `IldRequest::execute`, a second argument is passed. This argument constrains the number of rows to update to 1 despite the bind array size of 2. The last `select` statement checks that only one row has been updated.

Output Bindings

The `sbinding` example illustrates how to use:

- user-allocated memory to retrieve column data,
- internally-allocated memory for parameters,
- user-allocated memory for parameters.

It is built from the files `sbinding.cpp` and `ildutil.cpp`.

1. First, it connects to the database server and creates a table.
2. Then, input variables are used to insert rows in the newly created table. The prepared `insert` statement is used in repeated execution mode. The parameter values are passed to IBM ILOG DB Link, which assigns the necessary memory allocations.
3. The inserted rows are then fetched and the returned values are passed as the attributes of a user-defined object by binding the output columns via the function `IldRequest::bindCol`.

4. Finally, new rows are inserted using user-allocated memory for the parameter bindings. Then, a `select` statement is executed and the function `IldRequest::fetch` brings column data in the user object fields.

For both `select` statements, null indicator buffers are bound but are not checked at fetch time. This is not safe but the returned rows are known not to contain any null values.

Multiple Output Bindings

The `rebindcl` example illustrates the use of multiple bindings.

It is built from the files `rebindcl.cpp` and `ildutil.cpp`.

If the application needs to keep in memory the data retrieved from the RDBMS, it has to copy the data to its own internal buffers, since the buffer specified by the first `IldRequest::bindCol` operation will be overwritten by each successive `IldRequest::fetch` operation to record the newly retrieved data.

So, to avoid the overhead required to copy the data retrieved to another location, the `bindCol` function may be called between each `fetch` operation to specify a new memory area.

Handling LOBs

The `ildtext` and `ildbin` examples show how to handle LOBs (Large OBjects) with the use of the `IldRequest` functions `IldRequest::insertLongText`, `IldRequest::insertBinary`, `IldRequest::getColLongTextValue`, and `IldRequest::getLargeObject`. They are built from the files `ildbin.cpp`, `ildtext.cpp`, and `ildutil.cpp`.

These two samples take the data from two files whose names are given by the user, process the insertion into an ad-hoc table, and then retrieve the data into two new files.

The `clob` and `blob` examples show how to handle new CLOB and BLOB data types (reserved to Informix 9 and Oracle). These new data types are handled the same way as basic LOB types, so there is a common file between `ildtext` / `clob` (file `lobtext.cpp`) and `ildbin` / `blob` (file `lobbin.cpp`).

Asynchronous Processing

The `async` sample shows how the asynchronous feature may be used.

This feature is not implemented by every RDBMS. It may be used only against Mssql, Odbc (depending on driver capabilities), Oracle, and Sybase.

With the other RDBMSs, the sample will print a message to indicate that the feature is not implemented.

The sample will perform the following tasks:

- Set asynchronous status ON, and check that this worked correctly.
- Execute a simple insert and select operation.
- Run several queries simultaneously. This is to demonstrate that in asynchronous mode, when the application sends a request to the RDBMS, it gets the control back immediately, even if the RDBMS did not complete its task. Then, the application is free to do some other task. In this sample, we chose to submit other request to the RDBMS. Then, from time to time, the application has to check each request to ensure that it is completed.
- The cancel feature is also used to cancel a request too long to complete (run the sample with the '-c' parameter, (run the sample with no parameters for information on its usage)).

RDBMS-Specific Examples

A few RDBMS-specific examples are shipped with the standard distribution. They illustrate RDBMS-specific features.

- *Informix*

  For Informix9 or Informix US, the `inf9obj.cpp` sample file illustrates how to access `named row` and collection type columns.

- *Oracle*

  For Oracle, the `oraproc.cpp` file is an example of stored procedures while the `oracurs.cpp` file is an example of how to use an output parameter of cursor type.

  For Oracle, the `ora8obj.cpp` example illustrates how to access varrays and objects on selection, and how to use parameters of object and collection types.

  For Oracle, the `notif.cpp` example shows how an application can register to a list of events, and then get asynchronously notified when some of these events occur.

- *Sybase*

  For Sybase, you can find the following sample files:

  - `sybproc` for an example of a stored procedure call,
  - `sybtrig` for an example of a trigger firing,
  - `sybcomp` for an example of the `compute` clause.

Informix

SQL3 Features

The `inf9obj` example, built from the files `inf9obj.cpp` and `ildutil.cpp`, can only be run against an Informix Universal Server. Thus, the first argument passed to the `IldNewDbms` inline function is hard-coded.

In this example, a `distinct` type to be used in a collection and a `named row` type are created. These types are used to define two tables in which the following objects are inserted:

- A first row using a fully literal statement with an `IldObjectType` typed parameter,
- Then, other rows using a parameterized `insert` statement with an `IldCollectionType` typed parameter.

For the first object parameter insertion, the abstract data type descriptor is retrieved by its name using the function `IldDbms::getAbstractType`. However, this is not possible for the second insertion because the collection parameter needs an anonymous abstract type descriptor. This descriptor is retrieved via the table column description using the function `IldDbms::getRelation`. Then, the column abstract data type descriptor is accessed via the `IldDescriptor::getADTDescriptor` member function.

- Finally, two `select` statements are issued against the tables and the fetched rows are printed.

Stored Procedure Call

The `infproc` example is built from the files `infproc.cpp` and `ildutil.cpp`. Two procedures are created:

- The first procedure queries the catalog table `systables` for all table names and identifiers that match its second argument and whose owner is the one passed as its first argument. The matching table names and identifiers are returned as a standard result set that will be fetched.
- The second procedure sends an array of parameter values to be inserted in a temporary table.

Oracle

Stored Procedure Call

The `oraproc` example is built from the files `oraproc.cpp` and `ildutil.cpp`. It creates a `PL/SQL` package that contains a type definition and a procedure. Then, the procedure is called and the output values of the parameters are printed.

The procedure takes two parameters:

- The first one is a scalar integer and is used as an index for an array.
- The second one is an array that is modified by the procedure.

The parameter array size is mandatory since one of the parameters is of type array. Because the first parameter is a non-null scalar integer, its bind call does not need to use the optional arguments for the null indicator, the input/output status, or the actual array size. The user address of the value buffer is given but, because it is a fixed-size value type, the default value for size (-1) is passed: IBM ILOG DB Link will take care of the actual size.

For the second parameter, a specific array size is passed as the seventh argument in the bind call. That size is smaller than the maximum. Due to the procedure execution, it is clear that the actual value of the first parameter must be smaller than or equal to the actual array size of the second parameter.

The procedure sets some values for some elements of the array but it also sets an element to the null value, as can be seen when the returned array is printed.

The actual procedure call must be enclosed in an anonymous `PL/SQL` block.

The values of the parameters are retrieved using the IBM ILOG DB Link API but the user memory slots can also be accessed directly. The IBM ILOG DB Link API also tests whether the parameter values are null but this can have been checked by directly accessing the value of the user indicators that were bound using the `IldRequest::bindCol` function.

The first parameter does not require that a user-allocated memory space be bound.

The third argument to the binding of the second parameter is the actual user-side size of *one* element of the array.

The sixth argument to the binding of the second parameter is ignored by IBM ILOG DB Link for Oracle. Its value can therefore be set to `IlFalse` without any change in the execution behavior.

Cursor Output Parameter

The `oracurs` example is built from the files `oracurs.cpp` and `ildutil.cpp`. It creates a `PL/SQL` package that contains two type definitions and a procedure. Then, the procedure is called and the output values of the parameters are printed.

The procedure takes two parameters:

- The first one is a cursor that will be set during execution.
- The second one is a number used to restrict the `select` statement executed.

The sample first creates a table in which it inserts some rows before creating the package and calling the procedure. The returned value of the first parameter is then fetched just as with a usual `IldRequest` object that would have been used to execute a `select` statement.

This sample cannot be run against an Oracle server whose version is lower than 7.3.

Object Handling

The `ora8obj` example, built from the files `ora8obj.cpp` and `ildutil.cpp`, illustrates how the user-defined data-type features of Oracle are handled. It cannot be run against a server with a version lower than 8. This example is divided into three steps:

1. In the first step, an object type, a collection type, and tables with columns of those types are created. A parameter of `IldObjectType` type is used. Its value is built using the abstract data type descriptor returned from a call to the member function `IldDbms::getAbstractType`.
2. In the second step, rows are inserted using parameterized queries. A parameter of `IldCollectionType` type is used. Its value is also built using an abstract type descriptor.
3. In the third step, the contents of both tables are retrieved and printed.

Notification Sample

The `notif` example, built from the files `notif.cpp` and `ildutil.cpp`, demonstrates how the notification mechanism is implemented.

This feature is implemented only for the Oracle81 driver (this is a new feature in this database).

It is delivered with three SQL command files. These files need to be executed using SQL-Plus, for example. They achieve the following requirements:

- `notifocr.sql` for 'Notification objects creation'. This batch is to be executed first to create the queues required by the RDBMS to implement the notification mechanism.
- `notif.sql`. This batch will generate events that will be detected by the DBLink `notif` sample. It should be executed twice, when the DBLink sample is running.
- `notifodr.sql` for 'Notification objects drop'. This batch is to be executed once the DBLink sample is completed, to clean the queues and various objects created by `notifocr.sql`.

The `notif` sample demonstrates the following features:

- Subscribe to two different events: 'PUBSUB.INSERT_NOTIF:AGENT', and 'PUBSUB.UPDATE_NOTIF:AGENT'. When the subscription is done, a specific callback function is attached to these two events: 'insertCallBack' and 'updateCallBack'. Each callback function will display a specific message to show that it was called, and the insert callback will count the number of insert events.
- The sample will then wait for three insert operations. Note that the application may perform any operation during this time, and it is notified asynchronously when an event occurs. To simplify the sample, a sleep operation is done to wait for the events to be generated.

- Then, the sample will unsubscribe the update event. The same `notif.sql` batch is to be executed a second time to demonstrate that the update event is not received any more.

Sybase

Stored Procedure Call

The `sybproc` example is a free adaptation of the Sybase example `rpc.c`. It is built with the files `sybproc.cpp` and `ildutil.cpp`.

- It declares a stored procedure whose arguments are all output arguments, except the first one.
- A call to that procedure is then parsed, the parameters bound, and their values supplied.
- After execution, the various result sets returned are fetched in a double loop.
- Finally, the parameters output values are printed.

This sample shows the restrictions that exist when calling stored procedures against a Sybase server:

- Despite the fact that the statement must be prepared using the function `IldRequest::parse`, it cannot be executed several times.
- The `execute` SQL reserved word is mandatory. It is used as the only indicator of a procedure call during the parsing phase. This is one of the few cases where IBM ILOG DB Link needs to scan the query string,

If the procedure call only needs input parameters, you can simply use the `IldRequest::execute` function and pass the parameter values to the query string. For example:

```
request->execute("sp_helpdb mydb");
```

Another Sybase-specific feature with regard to stored procedure calls is that the output parameter values cannot be accessed before all result sets have been completely fetched.

Error Due to Trigger Firing

The `sybtrig` example is built from the files `sybtrig.cpp` and `ildutil.cpp`. It illustrates how to capture an error raised in a trigger fired by a `delete` event.

This example creates a table with a trigger attached to the `delete` event and inserts a row in the table such that it is protected against deletion by the trigger.

Then, it tries to delete the row, which causes the trigger to be fired. The trigger sends a TransactSQL `print` statement, rolls back the transaction, and raises a user-defined error.

The `print` statement is received as an information message and it is turned into a warning by IBM ILOG DB Link. The `raiserror` statement is actually returned as an error and is interpreted as such by IBM ILOG DB Link.

Compute Clauses

The `sybcomp` example, made of the files `sybcomp.cpp` and `ildutil.cpp`, illustrates the use of the Sybase `compute` clause.

This example:

- Creates a table,
- Inserts a few rows in this table,
- Issues a `select` statement that includes two `compute` clauses:
  - one for the minimum and maximum values of the column,
  - one for the average value of the column.

The three fetch loops show that after the normal result set containing the fetched rows, there is one result set for each individual `compute` clause.

Index

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W