



ILOG DB Link

EASY INTERFACE to RDBMS and ORDBMS from C++

White Paper



RELATIONAL AND OBJECT-RELATIONAL DATABASES CONNECTIVITY	3
MAIN BENEFITS	3
Simple interface	3
Fully Portable	3
Efficient Implementation	4
USING ILOG DB LINK	5
Kernel functionalities	5
RDBMS dedicated drivers	5
Sessions and connections	5
Query processing	5
SESSIONS AND CONNECTIONS	6
Single entry point	6
Connections handling	6
Complete transaction capabilities	6
Easy memory management	6
Cursor handling	7
METADATA ACCESS	8
Entity names	8
Various descriptors	8
Implicit descriptor management	8
QUERIES PROCESSING	9
Cursors and statements	9
Statements execution	9
Array modes	9
Bindings capabilities	9
Generic data types	10
Large Objects handling	10
Data accesses	10
ERRORS HANDLING	11
Mechanism	11
Diagnostics	11
Customizing	11
CONCLUSION	11
EXAMPLES	12
Bindings	12



Relational and Object-relational databases connectivity

ILOG DB Link packages all the features of various RDBMS through a simplified C++ interface. ILOG DB Link easily handles simultaneous connections to one or more databases in one or more RDBMS.

By simply adding a few lines of code, your application will retrieve and modify data from one or more databases using an interface that is independent from the RDBMS itself. Furthermore, in addition to the simple manipulation of data in your databases, ILOG DB Link also enables you to access your database metadata, execute stored procedures and manage sessions using C++ objects.

ILOG DB Link is a set of C++ libraries that enables you to make full use of all the services provided by your relational database management system (RDBMS) such as Informix™, Ingres™, Microsoft SQL Server™, Oracle™, Centura SQL Base™, Sybase™ or ODBC compatible RDBMS.

Moreover, as C++ libraries, it is easily integrated into your own applications. ILOG DB Link provides your applications with code portability and RDBMS Application Programming Interfaces (API) independence: the application can be targeted towards a specific RDBMS or be build independently of the RDBMSs to which it will connect at runtime.

Main benefits

If your C++ applications need persistent storage of data, or access to data stored in your firm's databases, you will need fast and straightforward access libraries. Although several database class libraries are available, ILOG DB Link provides simpler and faster accesses to your corporate data.

This white paper will describe the following benefits of ILOG DB Link:

- high-performance RDBMS access, including large objects (LOBs)
- multithreaded sessions to multiple databases
- transparent usage of SQL including DBMS-specific extensions.

These benefits are due to ILOG DB Link *simple interface* which allow to use it in a matter of minutes, its *fully portable* across RDBMS ad operating systems API and its *efficient implementation* which minimizes the number of functions to be used reducing the database accesses code while allowing a maximum independence from the context and minimizing the network load.

Simple interface

Complete access to any RDBMS is implemented in ILOG DB Link using only three classes which encapsulate:

- the current connection and all schema handling capabilities,
- all input and output data handling and query processing,
- the error handling mechanism.

More classes are available for advanced applications needing an in depth knowledge of the database metadata. These classes implement descriptors of the schema entities like:

- tables and views,
- stored procedures and functions,
- synonyms,
- and abstract, or user defined, data types.

Moreover, ILOG DB Link API is an object variant of the Call Level Interface API defined by the ISO SQL standard.

Fully Portable

ILOG DB Link has a uniform API thereby ensuring that your application is portable across different RDBMS and across different operating systems.



DB Link White Paper

You do not need to modify your application code in order to fit the target RDBMS, it can even not be targeted to any RDBMS using the *dynamic driver load* feature: it will only be linked with the driver manager which will take care, at runtime, of loading the appropriate RDBMS driver.

However, as ILOG DB Link transmits your SQL queries in the form of unmodified character strings, you can still use the SQL enhancements specific to your RDBMS such as the Sybase "compute" clause or the Oracle "savepoint" clause.

Efficient Implementation

ILOG DB Link offers the abilities to handle data to be sent or retrieved by arrays of rows rather than one row at a time: this is achieved using the *array modes* which are turned on to the desired number of rows to be processed with only one function call. These features augment the application throughput by reducing, every time it is supported by the RDBM client API, the network traffic.

ILOG DB Link handles all necessary bindings for the current query leaving the developers free to process only the one needed by the application design.

ILOG DB Link provides utility classes to gain independence from the RDBM server and client settings as well as from the locale language settings for date and time data as well as large exact numeric data.

Moreover, when communicating with an ORDBMS, the developers can make use of a specific class which can hold values of any abstract or user defined data type.



Using ILOG DB Link

ILOG DB Link is built using a two-level model. The first level contains the kernel functionalities which are used by the application developers, and the second level implements the RDBMS dedicated drivers. The main capabilities of ILOG DB Link kernel are implemented in two broad classes : the session and connection to the RDBMS form the first class, and the second class handles the processing of queries.

Kernel functionalities

Applications developers only need to know about the ILOG DB Link kernel functions thus will not have to worry about the specific RDBMS API and processing protocol.

Most ILOG DB Link functions uses several functions from the RDBMS API and thus makes the application faster and easier to develop.

RDBMS dedicated drivers

ILOG DB Link drivers can be either statically linked into the application executable or dynamically loaded at runtime.

From the application developer point of view, they are libraries to be used and nothing more.

When the application is designed for dynamic loading, the libraries will be distributed along with the application.

Sessions and connections

Instances of the class `IldDbms` are equivalent to open sessions and embody the different connections to a RDBMS. One `IldDbms` object is usually connected to a single database situated in a particular RDBMS. However, during a given session, a `IldDbms` object can be repeatedly disconnected and reconnected to and from the same database. Furthermore, within the same RDBMS, it can even be disconnected from one database and connected to another.

Query processing

Instances of the class `IldRequest` are used to parallel the cursors in a RDBMS but are designed so that they handle as well non cursory statements.

As a cursor is directly related to a specific connection, the connection is responsible for the cursor management.



Sessions and connections

Single entry point

Sessions are only created by the function `IlDNewDbms` and embody the current connection to a particular database. Actually this function is the only entry point to ILOG DB Link. It controls that a driver for the specified RDBMS is present or, when using the dynamic load feature, tries to load into memory the required library.

Then this function call the `IlDDBms` constructor function in which an initial connection will be established. ILOG DB Link has no intrinsic limitation upon the maximum number of connections that can be opened simultaneously, if such a limitation exists it is due to the RDBMS configuration.

When looking for a driver to load, ILOG DB Link will search the configuration file to get the driver library name and use the environment variables to search the library.

Connections handling

When a session is created, an initial connection is established with the database server. This is a communication channel that can be closed and reopened on demand. The connection can be either established or closed and the application can check at all times its status.

To connect the application to a database, you give a connection, or authentication, string argument that varies according to the target RDBMS.

When the application is disconnected, all objects directly related to it are destroyed i.e., all the cursors and all the pooled schema entity descriptors are deleted.

At the connection level, apart from the login information like the user and the database names, applications have access to the values of the configuration options and the features usage.

Other available informations are the server informations like its transaction capabilities, the maximum number of tables that can be joined in a select statement, the maximum number of columns a table can have, ...

Complete transaction capabilities

Although transaction capabilities handling vary according to the target RDBMS, ILOG DB Link provides a unified interface which encompasses all the various features necessary for transaction management.

ILOG DB Link includes the necessary API to fully support transaction handling : the application can start a transaction, optionally named with RDBMS which support that feature, process the SQL statements pertaining to the transaction and then commit or roll back the transaction.

Using this API, it can also modify the auto-commit mode of the connection when the RDBMS supports it.

All the functions relating to transaction handling are implemented for all supported RDBMS despite the features are not always supported : ILOG DB Link will silently ignore the ineffective functions.

To meet the specific requirements of a particular RDBMS, all these functions have optional parameters which can be used freely. When they are not required, they are simply ignored by ILOG DB Link.

Easy memory management

Destroying a session object and closes the current database connection and it also frees or de-allocates all objects associated with it. In this way, despite the fact the destructors of these classes are public, the



DB Link White Paper

application is not concerned about the memory management of the related objects. The associated objects are the opened cursors, and the schema entity descriptors that were kept in the cache.

Cursor handling

New cursors are only be allocated at the connection level. A cursor is an instance of the class `IldRequest` and can only be obtained from an existing opened connection. The cursors are managed by the connection they are attached to and are by pooled in the connection cache. The cursors are pooled in a cache hold by the connection object. Once the application does not need a cursor any more it can return it to the pool or actually destroy it.



Metadata access

With ILOG DB Link, the database metadata is known as a set of schema entities which can be described using descriptor objects.

A schema entity is any autonomous structure in a schema: this includes the tables, the views, the stored procedures, the user defined data types and the synonyms. On the opposite, indexes and primary keys are not considered as schema entities because they can not be described independently from a table.

Entity names

ILOG DB Link allows applications to retrieve the entity names and owner names for any category of schema entity.

Various descriptors

Basically, any schema entity descriptor has a name, an owner, a related connection, and, on most RDBMS, a numerical identifier.

The schema metadata is described using 4 main classes: one for tables and views, one for stored procedures and functions, one for synonyms and one for user defined data types.

From a table description, an application can query the existence of a primary key, foreign keys and attached indexes.

A procedure or function descriptor can tell the number of arguments needed and the number of resulting values.

Each argument to the procedure or function is described by the same characteristics as a relation column and has also an input or output status and an indication of the existence of a default value.

An abstract data type descriptor knows if it describes a collection, like an array or a list, or an object or a named row.

Also attached to the descriptor are the attribute descriptors as well as the method descriptors.

Each attribute descriptor is of the same class as a relations column descriptor.

Each method descriptor is a procedure or function descriptor.

Implicit descriptor management

When querying the RDBMS for a schema entity, the returned description can be cached in the connection and thus ILOG DB Link will provide the fastest possible access to that descriptor in further calls but will also take care of the related memory management.

On the opposite, applications can get these descriptions without using the cache and then are responsible for the memory de-allocation.

When a descriptor is explicitly destroyed, the cache will be automatically updated if the descriptor was registered.



Queries processing

Cursors and statements

A single class handles cursory statements, that is select statements or stored procedure calls, as well as non cursory statements, that is Data Manipulation Language (DML) statements, like insert or update statements, and Data Definition Language statements, like create and drop statements.

Instances of that class are pooled by the connection object. If it is designed in such a way, an application can explicitly delete a cursor but returning a cursor to the pool allows an application to have a faster access to such objects while it does not need to care for the memory usage.

Statements execution

A SQL statement can be executed directly or be first prepared for execution if it contains place holders : ILOG DB Link supports both modes. While input parameters must be bound before execution, the select list columns can be bound only before fetch.

ILOG DB Link maximal transparency with respect to the SQL statements joined to the fact most RDBMS do not describe the parameters impose that all parameter bindings be processed by the application but the memory management can be left to ILOG DB Link.

On the opposite, ILOG DB Link let the application bind or not the select list columns either to the application proper memory space or will automatically allocate the required memory. ILOG DB Link provides specialized accessors to retrieve the columns data when the results set columns were not bound.

After execution, the application will submit a fetch command to the RDBMS. When the list of tuples selected by the most recent select command is exhausted, more calls for fetch have no effect. ILOG DB Link also supports multiple results sets returned from the execution of a single SQL statement like it is the case with Sybase or Microsoft SQL Server execution of a stored procedure call: with ILOG DB Link the application will just keep fetching after no data were returned and the next results set will be brought back.

The SQL standard defines the question mark "?" as the placeholder mark format but, when using Oracle or Centura SQL Base, the placeholders can be given using a name or a number:

```
select * from atable where acolumn = :acol  
or  
select * from atable where acolumn = :1
```

Both formats are supported by ILOG DB Link.

Array modes

ILOG DB Link allows bindings to take place over arrays of rows. This is called "array fetch" for output bindings and "array bind" for input bindings. The array mode can be turned on or off, independently, for select-list columns and input parameters. The application sets the number of rows to be fetched or the number of rows of variables to be sent at once.

On RDBMS which do not support array bindings, ILOG DB Link emulates the required mode so that application codes do not need to be changed but they will not benefit from the increase in throughput because the network load will not be lowered.

Bindings capabilities

Binding an output column informs the RDBMS API that after being converted to the appropriate type, data must be fetched into the area pointed to by a given address. If the application changes the output data type, it will be dependent upon the RDBMS support for the conversion unless the column datatype is one which ILOG DB Link will convert to one of the generic types it supports.



Binding an input parameter informs the RDBMS API that the expected values are to be taken at a certain address and are of a designated type. This binding can also give the actual number of values in the data buffer when calling an Oracle stored procedure whose arguments are of array types.

Generic data types

- Dates and times

In its default configuration, ILOG DB Link sends and retrieves date values in character string form. You must therefore be careful when using the currently active date and time format in the application which must process the data according to the externally defined format. This default configuration is called the "date as string" feature and can be turned off, enabling you to send and retrieve date and time values as objects.

- Numerically exact

In financial applications, you usually need exact numerical precision in order not to lose any significant digits when handling data. With really big numbers this cannot be guaranteed if the values are mapped to the host language integer or floating point numbers. ILOG DB Link offers the "numeric as string" feature which allows you to send and retrieve exact precision numbers in string form.

Still the application has a dependence upon the local languages: some use a dot as decimal part marker, others use a coma. To overcome this, ILOG DB Link offers the "numeric as objects" feature which allow applications to send or retrieve numeric values as instances of the class `ILDNumeric`.

- Abstract data types

When connected to ORDBMS, applications need to send and retrieve data of unknown structured types like arrays, lists or objects. ILOG DB Link has the ability to do so using instances of a class which allows to handle 'horizontal', like objects and named rows, as well as 'vertical' structures like arrays and sets.

Like with any other datatypes, values for abstract data types can be bound on input or output.

Large Objects handling

Multimedia applications are now very common and make a huge use of RDBMS stored large objects such as long text or images. ILOG DB Link offers a unified interface with which you can send or retrieve LOB data as you do for any other data type. LOBs can be retrieved either:

- in ILOG DB Link allocated memory space, with a restriction on the maximum data size
- in application allocated memory space, with the maximum size restricted by the application
- into a named file, with no size restriction
- by chunks into application allocated memory, with the application setting the size of each chunk.

Because some database systems restrict the use of large objects (LOBs) in queries, ILOG DB Link offers a specific API to send and retrieve LOB data to and from the server.

ILOG DB Link also allows, whenever possible, to retrieve large objects by pieces. The application can then start a query which will return the data from a LOB column and fetch it by pieces of whatever size it choose.

Data accesses

On output as well as on input, ILOG DB Link maps the RDBMS data types to C++ types which are identified by the values of an enumeration. To one value from the enumeration corresponds a C++ type into and from which the data will be converted.

Not all types always map an actual column data type when connecting to a specific RDBMS, but it is guaranteed that the minimal ANSI set of data types are present: integer, string, real and date types exist for all RDBMS.

The data selected while a select command is run, is returned to the application in the form of tuples stacked in a cursor.

ILOG DB Link provides full descriptions of the select list columns by means of descriptors.



To retrieve column data values, there is a specialized accessor for each ILOG DB Link data type, an error is raised if the actual column data type does not match the one for which the accessor is designed.

Errors handling

When using a query language in a client/server configuration, there are many reasons why error conditions are raised. An error can either be raised by ILOG DB Link or by the RDBMS. There are also messages sent back by some of the database servers or user messages provoked by specific SQL statements.

Mechanism

ILOG DB Link error handling mechanism relies on an error reporter class which can be derived to enforce an application-specific error handling protocol. An error reporter exists as soon as a connection or a cursor is created but it can be changed to fit the application security policy needs.

When an error condition is met, ILOG DB Link uses the error reporter of the current invoking object. This error reporter saves the error context in an instance of the class `IldDiagnostic` and then calls one of two functions depending of the layer the error originated from : ILOG DB Link API or the database server. The actions of these functions can correct the error and then rerun the failed command.

This schema makes that only one error at a time can be raised within ILOG DB Link.

Diagnostics

ILOG DB Link offers a single class as interface to both types of messages : errors and warnings. Its instances contain informations about the context in which the error, or warning, was raised like the error code, the error message text and the name of the function in which the message originated.

When a function from ILOG DB Link API is called, the caller object must be tested for error conditions. It can also be tested for warnings.

Applications can gain access to diagnostic instances as a whole or only reach at the individual slots.

Customizing

The error reporter can be the same for all connection and cursor objects. At creation time, instances of these classes have a default reporter posted, this default reporter is not accessible to the application and will be reset as current reporter whenever the application defined reporter is removed or deleted. The reporter objects have the particularity to be automatically deleted when their last reference is removed.

When customizing the error handling mechanism, two functions must be redefined in order to fit the application error protocol. The first one will be called whenever the database server returns an error during a query processing while the second will be called whenever ILOG DB Link detects an incorrect parameter value to any interface function.

Conclusion

ILOG DB Link provides all the interface functions needed to integrate a RDBMS with a C++ application. Not only does it save the user from writing very tedious code, it provides both portability across platforms and database vendor independence while it still leaves the applications use RDBMS specific extension to SQL.

ILOG DB Link not only simplifies application development but it also unifies the communications with RDBMS by resorting to emulation when a feature is not supported by the server or the client.



Examples

Bindings

In this example, you will discover the different binding modes offered by ILOG DB Link.

Input variables are declared but the memory management is left to ILOG DB Link while an application-defined class is used to retrieve data using output bindings.

The parameters are bound using ILOG DB Link types and the first one is given a maximum size but the memory allocations management is left to ILOG DB Link. Array bind mode is activated thus parameter rows are all sent using a single execute call.

Output bindings are done using application-allocated memory spaces. The first output column is bound to a character string type variable and the second to an integer type variable. The maximum available string length is given on the first binding; it is useless for the second and is not given.

For both columns, no null indicator buffer is given while this is harmless since ILOG DB Link will allocate the required buffers. The application should usually check the nullness of the column before accessing the potentially returned data. Column data is automatically placed in the bound memory spaces during the fetch. Since array fetch is not used, the application fetches only one row at a time, and the same positions are overwritten by the successive calls.

Finally, on exit, only the connection object is deleted because it will take care of the cursor disposal.

```
class Customer {
public:
    Customer() {} ;
    ~Customer() {} ;
    char name[21];
    IldInt age;
};

int main(int argc, char** argv)
{
    Customer cust ;
    IldDbms* dbms = IldNewDbms(argv[1], argv[2]);
    if (!dbms || dbms->isErrorRaised())
        exit (1);
    IldRequest* request = dbms->getFreeRequest();
    If (dbms->isErrorRaised())
        IldDisplayError("No cursor available", dbms);
    request->setParamArraySize(5);
    if (!request->parse("insert into BINDTABLE (NAME , AGE) values ( ? , ?)")
        IldDisplayError("Parse failed:", request);
    if (!request->bindParam(IldUShort)0, IldStringType, 20))
        IldDisplayError("Binding of parameter 1 failed:", request);
    if (!request->bindParam(1, IldIntegerType))
        IldDisplayError("Binding of var2 failed:", request);
    char* names[5] = {"Birgit", "Ann", "Bill", "Francoise", "Julio"};
    IldInt ages[5] = {23, 5, 66, 43, 50};
    for (int i = 0; i < 5; i++) {
        request->putVarValue(names[i], 0);
        request->putVarValue(ages[i], 1);
    }
    if (!request->execute())
        IldDisplayError("Execution failed:", request); }
    if (!request->execute("select NAME, AGE from BINDTABLE"))
        IldDisplayError("Execution failed:", request);

    if (!request->bindCol((IldUShort)0, IldStringType, cust->name, 20))
        IldDisplayError("Binding column 1 failed:", request);
    if (!request->bindCol(1, IldIntegerType, &(cust->age)))
        IldDisplayError("Binding column 2 failed:", request);
    while (request->fetch().hasTuple())
        cout <<" Customer : " << cust->name << " " << cust->age << endl;
    if (request->isErrorRaised())
```



```
IldDisplayError("Fetch failed: ", request);
delete dbms;
return 0;
}
```

Generic types

In this example, a table is created with column data types whose data can be retrieved using the generic data types of ILOG DB Link: date and exact numeric types. The column types are built by two utility functions which discriminate upon the RDBMS name to determine which is the appropriate SQL type name.

Rows are inserted in the table using parameter bindings. The first parameter refers to the exact numerical data type column, the second to the date data type column. Here SQL standard place holder markers are used.

While the first column is of a numeric type, NUMERIC or DECIMAL depending on the target RDBMS, the first parameter is bound to string type because the conversion is supported by the RDBMS.

Huge numbers that cannot be represented exactly in machine number format are inserted along with dates defined through `IldDateTime` instances.

Eventually, data is retrieved using the numeric as object feature for the first column and the date as object feature for the second.

```
int main(int argc, char** argv)
{
    IldDbms* dbms = IldNewDbms(argv[1], argv[2]);
    if (!dbms || dbms->isErrorRaised())
        exit (1);
    char createStr[80];
    ostrstream ostr(createStr, 80);
    ostr << "create table ATABLE(F1 " << IldGetNumericTypeName(argv[1])
        << ", F2 " << IldGetDateTypeName(argv[1]) << ")" << ends;
    if (!request->execute(createStr)) {
        IldDisplayError("Table creation failed: ", request);
        delete dbms;
        exit(1);
    }
    // Use DateTime structures
    request->setStringDateUse(IldFalse);
    // Data insertion.
    if (!request->parse("insert into ATABLE values(?, ?)") {
        IldDisplayError("Parse of insert failed:", request);
        Ending(dbms, request);
    }
    if (!request->bindParam((IldUShort)0, IldStringType)) {
        IldDisplayError("First variable binding failed:", request);
        Ending(dbms, request);
    }
    if (!request->bindParam((IldUShort)1, IldDateTimeType)) {
        IldDisplayError("Second variable binding failed:", request);
        Ending(dbms, request);
    }
    IldInt nbRow = 0;
    request->putVarValue("1234567890123.4567890", 0);
    IldDateTime* dt = new IldDateTime(1996, 3, 2); // 1996/03/02
    request->putVarValue(dt, 1);
    if (!request->execute(&nbRow)) {
        IldDisplayError("First insertion failed: ", request);
        Ending(dbms, request);
    }
    request->putVarValue("9876543210987.654321098", 0);
    dt->setMonth(1); // 1996/01/25
    dt->setDay(25);
    request->putVarValue(dt, 1);
    if (!request->execute(&nbRow)) {
```



```
IldDisplayError("Second insertion failed: ", request);
Ending(dbms, request);
}
request->putVarValue("9876543210012", 0);
dt->setMonth(2); // 1996/02/12 14:33:44.000
dt->setDay(12);
dt->setHour(14);
dt->setMinute(33);
dt->setSecond(44);
request->putVarValue(dt, 1);
if (!request->execute(&nbRow)) {
    IldDisplayError("Third insertion failed: ", request);
    Ending(dbms, request);
}
delete dt;

request->setNumericUse(IldTrue);
if (!request->execute("select * from ATABLE")) {
    IldDisplayError("Select failed: ", request);
    Ending(dbms, request);
}
// Print selected item names.
cout << "ATABLE" << endl
    << request->getColName(0) << "\t" << request->getColName(1) << endl;
// Print selected item values.
do {
    if (!request->fetch())
        IldDisplayError("Fetch failed:", request);
    else if (request->hasTuple()) {
        IldNumeric num = request->getColNumericValue(0);
        if (request->isErrorRaised()) {
            IldDisplayError("Numeric get failed: ");
            break;
        }
        else {
            char numValue[ILD_MAX_NUM_LEN];
            num.get(numValue, ILD_MAX_NUM_LEN);
            cout << numValue << "\t";
        }
        IldDateTime dt = request->getColDateTimeValue(1);
        if (request->isErrorRaised()) {
            IldDisplayError("Cannot retrieve DateTime: ", request);
            break;
        }
        else
            cout << dt.getYear() << "/" << dt.getMonth() << "/"
                << dt.getDay() << " " << dt.getHour() << ":"
                << dt.getMinute() << ":" << dt.getSecond() << endl;
    }
} while (request->hasTuple());
```