IBM ILOG DB Link V5.3

Tutorial

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*>/license/notices.txt in the installed product.

Table of Contents

C H A P T E R

1

IBM ILOG DB Link Tutorial

IBM® ILOG® DB Link is a comprehensive C++ library that handles the processing of Relational Database Management Systems (RDBMS). It includes several classes of objects that allow efficient development of applications with RDBMS connectivity. The API is simplified to hide the complexity of the Client API of the various RDBMSs. Furthermore, the DB Link API is the same for any RDBMS. Consequently, applications developed with the DB Link library will work with Oracle®, Sybase, or Informix (for example) without any change in the source code.

A schematic representation of the IBM ILOG DB Link Architecture is shown in Figure 1.1.

This tutorial presents the main features of the library through the use of samples. Each significant point is detailed with the corresponding excerpt from the source code. The full code is also provided, if you wish to get a complete view of the mechanism.

The tutorial is divided in 4 chapters:

- ***IBM ILOG DB Link Basic Use*** - Describes the basic features of DB Link—how to connect and execute queries. It explains the fundamental classes of IBM ILOG DB Link: `IldDbms`, `IldRequest`, and `IldDiagnostic`.
- ***IBM ILOG DB Link Optimization Techniques*** - Describes the methods used to optimize an application when using DB Link to run queries on an RDBMS.
- ***IBM ILOG DB Link Portability Considerations*** - Describes the special considerations to keep in mind when building portable applications.

- ***IBM ILOG DB Link Access to Object Data Types*** - Presents how to access the new data types introduced by Object Oriented RDBMS.

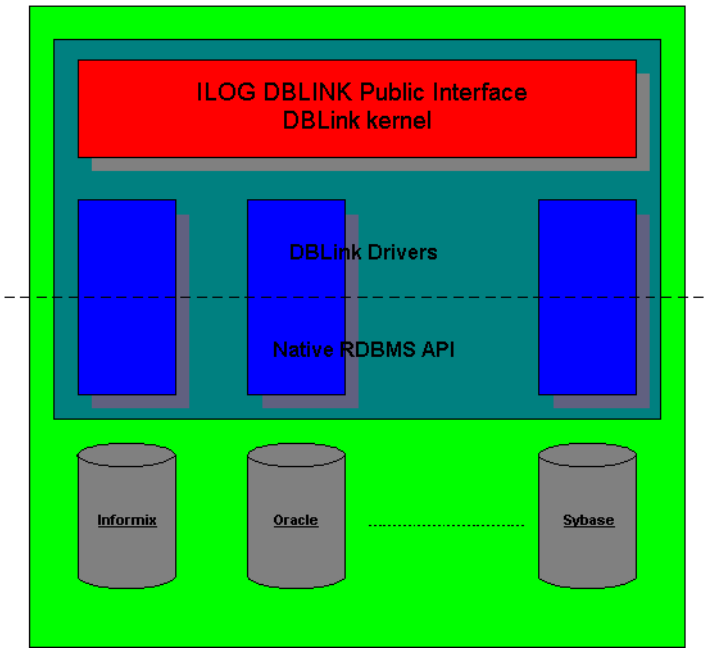

Figure 1.1 *IBM ILOG DB Link Architecture*

IBM ILOG DB Link Basic Use

This chapter details the fundamental principles of IBM® ILOG® DB Link through the use of a sample. Working through the 4 steps found in this chapter will help you understand how to send basic queries to the database server and then retrieve the results.

The main class is `IldDbms`, which handles the connection to the RDBMS. It also gives access to the schema handling capabilities and manages various configuration settings.

Then, the `IldRequest` class is designed to send queries to the RDBMS and get the results back. To send a query, you first need to be connected. Therefore, instances of `IldRequest` are created or released through an `IldDbms` instance that handles the connection.

The `IldDiagnostic` class is used to manage errors that may be raised. These errors can come from the RDBMS or from DB Link itself.

The `IldDbms` and `IldRequest` classes get access to the error information by the same API. This reduces the time required to learn how to handle the errors and is achieved with the `IldIldBase` class.

`IldIldBase` is an abstract class: you do not need to create an instance of this class. Its only purpose is to implement the error mechanism that will be used by the `IldDbms` and `IldRequest` classes. They both inherit from `IldIldBase`. `IldIldBase` includes an instance of the `IldDiagnostic` class and provides methods to access the error information.

This tutorial has 4 steps:

- ***Step 1: Connecting to the Database*** - Shows how to connect to the database, using the IldDbms class.
- ***Step 2: Querying the Database*** - Sends a query to the database.
- ***Step 3: Retrieving Data from the Database*** - Retrieves the output from a *select* query.
- ***Step 4: Disconnecting From the Database*** - Disconnects from the database and releases the objects.

Step 1: Connecting to the Database

This step shows how to connect to the RDBMS and disconnect when leaving the application.

The code is presented, beginning with the **main** part:

```
IldDbms* dbms = IldNewDbms(dbName, connStr) ;
```

This is the only entry point to the library. A single entry point simplifies the use of the library. Using the `IldNewDbms` method is all that is required to get connected, and is the only way to get a connection. Any object used later will be allocated from the `IldDbms` instance returned by this method. Therefore, destroying the `IldDbms` instance will also automatically release all objects allocated with this instance.

This method has 2 arguments. The first identifies the RDBMS to connect to. There is a specific name for each RDBMS supported by DB Link. The second argument specifies the user name, password, and database identification. Both these arguments are given as parameters to the program.

The connection string format depends on the RDBMS, as described in the following table:

dbName	connStr
informix	userName/password/database@dbServer
mssql	userName/password/database/dbServer
odbc	dataSourceName/userName/password

dbName	connStr
OLE DB	userName/password/database/dbServer
oracle	userName/password@service
sybase	userName/password/database/dbServer

It is possible for the connection to fail. This is the case, for instance, when the user password is incorrect. The following code tests for such a failure:

```
if (dbms->isErrorRaised()) {
  IldDisplayError("Connection failed : ", dbms) ;
  delete dbms ;
  exit(1) ;
}
```

The `IldDbms` class can indicate whether an error occurred and what kind of error it was. The `IldDisplayError` function queries the `IldDbms` instance to get this information and then process it (display it in this case). This is done as follows:

```
void IldDisplayError(const char* operation, const IldIldBase* ildobj) {
  cout << operation << endl;
  cout << " Code : " << ildobj->getErrorCode() << endl;
  cout << " SqlState: " << ildobj->getErrorSqlstate() << endl;
  cout << " Message : " << ildobj->getErrorMessage() << endl;
}
```

Before exiting the program, the `IldDbms` object must be deleted. Doing so automatically disconnects from the database and releases any objects previously allocated with this connection. This is done very simply, as follows:

```
delete dbms;
```

Conclusion

This chapter showed how to connect to an RDBMS and process the errors that may occur. You are now ready to send queries to the server.

See source code.

Step 2: Querying the Database

To send a query to the database, you need to get an instance of the `IldRequest` class. This class is used both to send queries to the RDBMS and to retrieve results. The instance is created using the `IldDbms::getFreeRequest` method.

After connecting to the RDBMS as described in Step 1, a new request is allocated from the connection (this request will be used later to send queries to the RDBMS). This is done as follows:

```
IldRequest* request = dbms->getFreeRequest() ;
```

```
if (dbms->isErrorRaised()) {
  IldDisplayError("Creation of request failed : ", dbms) ;
  delete dbms ;
  exit(1) ;
}
```

> ***Note:*** *The error handling mechanism is the same as the one used with the* `IldDbms` *instance in Step 1.*

Then, the table is created by using this new `IldRequest` instance to send a DDL (Data Definition Language) statement to the RDBMS. This is done using the `IldRequest` method `execute (const char*, IldInt* rowCount = 0)`, as follows:

```
const char* createStr = "create table ATABLE(F1 int,F2 char(20))" ;
cout << "Creating a table : " << createStr << endl ;
request->execute(createStr) ;
if (request->isErrorRaised()) {
  IldDisplayError("Table creation failed : ", request) ;
  delete dbms ;
  exit(1) ;
}
```

The second argument of the `execute` method is an optional output argument. It is used to get the number of rows modified by the statement. It is not applicable for a DDL statement but it can be used for a DML (Data Manipulation Language) statement, such as *insert*.

An *insert* statement now writes records to the table (this is the simplest way to insert a row in a table).

```
const char* insertStr1 = "insert into ATABLE values(40,'Forty')" ;
IldInt nbRows = 0 ;
cout << "Row #1 : " << insertStr1 << endl ;
if (!request->execute(insertStr1, &nbRows))
  IldDisplayError("Insertion failed :", request) ;
else cout << "\t" << nbRows << " row inserted." << endl ;
```

The number of inserted rows is retrieved with the second parameter of the `IldRequest::execute` method. This is useful mainly when running an *update* statement together with a *where* clause. In such a case, you may not know how many rows are updated. The only way to be made aware of this is to use this second parameter.

This time the error was checked using the '!' unary operator. It is redefined by the `IldIldBase` class to return the error status. Therefore, it may be used with either an `IldDbms` or an `IldRequest` instance. It is equivalent to the `isErrorRaised` method, except that it is shorter to write.

Before leaving the program, the table that has just created must be cleaned out. This is done the same way it was created— by using a *drop* statement.

```
const char* dropStr = "drop table ATABLE" ;
```

```
cout << "Dropping table : " << dropStr << endl ;
if (!request->execute(dropStr))
  IldDisplayError("Drop table failed : ", request) ;
```

Conclusion

These instructions are all that is needed to send a query to the RDBMS. They will be used often.

See source code.

Step 3: Retrieving Data from the Database

This step shows how to retrieve the output from the database. In Step 2, a simple query to create a new table was sent to the RDBMS, and records were inserted in this table. Now, you can read the data from the table.

First, a *select* statement is executed. This is done as follows:

```
const char* selectStr = "select * from ATABLE";
cout << "Retrieving all rows : " << selectStr << endl;
if (!request->execute(selectStr))
  IldDisplayError("Select failed : ", request);
```

> ***Note:*** *The value of* `rowCount` *passed to* `execute` *is* `0` *for "select" queries. This is logical—counting the rows before they have all been accessed would impose an unnecessary performance penalty on the application.*

When `execute` is successful, a description of the results set can be accessed. The description includes the number, name, data type, and size of each column. These are accessed by using the following methods:

- `IldUShort IldRequest::getColCount` - Gets the number of columns in the results set.
- `const char* getColName(IldUShort index)` - Gets the name of the column defined by the index position in the results set.
- `IldColumnType IldRequest::getColType(IldUShort index)` - Gets the type of the given column.
- `IldUInt IldRequest::getColSize(IldUShort index)` - Gets the size of the column (number of bytes required to store a value of the column).

In the sample, you know exactly what results set to retrieve. However, these methods are used to display the results set when the query is known only at run time. This is *Dynamic SQL;* the application can process queries provided by an end user at run time.

The information is processed as follows:

```
// Print selected item names.
IldUShort i ;
IldUShort nbCols = request->getColCount() ;
const char* colName1 = request->getColName(0) ;
const char* colName2 = request->getColName(1) ;
cout << "\t ATABLE" << endl ;
cout << " " << colName1 << "\t\t" << colName(2) << endl ;
```

The information is available following the execution of the *select* query. `IldRequest::fetch` is now used to get the values of these columns from the RDBMS. This retrieves the data from the memory of DB Link.

`IldRequest::fetch` attempts to get the first available row of the results set. If there is a row available, the `IldRequest::hasTuple` method returns `IldTrue`.

Since `IldRequest::fetch` returns a reference to an `IldRequest` instance, the rows are fetched in a loop as follows:

```
while (request->fetch().hasTuple()) {
  cout << " " ;
  for (i = 0 ; i < nbCols ; ++i) {
    if (request->isColNull(i))
      cout << "-" ;
    else
      switch (request->getColType(i)) {
      case IldIntegerType :
        cout << request->getColIntegerValue(i) ;
        break ;
      case IldStringType :
        cout << "'" << request->getColStringValue(i) << "'" ;
        break ;
      default :
        // Other possible types are not handled here.
        break ;
    }
    cout << endl ;
  }
  cout << endl ;
}
```

As long as there is a tuple available, one of the `IldRequest::getCol<dataType>Value(IldUShort i)` methods is used to retrieve the values of the results set. The method depends on the column type. The IBM ILOG DB Link Reference Manual contains a complete list of these methods.

> ***Note:*** *A column of any type may be* `null`*. This is detected by the method* `IldRequest::isColNull(IldUShort i)`.

Conclusion

This step described the simplest way to read the data recorded in the database. This includes the *Metadata* of the results set, as well as the data itself. You can now run queries against the database. Since your queries will probably be more sophisticated than these, the sample code

can be changed to improve performance. Nevertheless, the basic approach will be the same for all DB Link applications.

See source code.

Step 4: Disconnecting From the Database

This step shows how to terminate the program by disconnecting from the database and releasing the objects previously allocated.

The simplest way to do this is to delete the `IldDbms` object itself. This automatically disconnects from the database and releases any objects previously allocated with this connection. This is done as follows:

```
delete dbms;
```

Conclusion

This step demonstrated the disconnection procedure. This procedure is simplified since an `IldDbms` instance keeps track of the objects allocated with its connection.

IBM ILOG DB Link Optimization Techniques

This chapter describes 4 techniques that can be used to optimize an application when working with an RDBMS.

- ***Step 1: Executing a Query Multiple Times***, shows the proper way to execute a query several times.
- ***Step 2: Optimizing Network Use***, shows how to optimize use of the network.
- ***Step 3: Accessing Data Directly***, shows how to optimize data transfer with the RDBMS.
- ***Step 4: Keeping Data in Memory***, shows how to efficiently keep data read from the RDBMS in memory.

Step 1: Executing a Query Multiple Times

This first optimization technique is for cases when a query has to be executed several times. This technique involves the use of *deferred execution*. With deferred execution, the statement is first prepared for execution and then executed. With *immediate execution* (as seen in *IBM ILOG DB Link Basic Use*) the two steps are carried out simultaneously. Deferred execution is used in the following cases:

- ***The Statement is to be Executed Several Times***: In this case, the statement is prepared once by the server for all the required executions. This eliminates the preparation time for the subsequent executions.

- ***The Statement Contains Parameters***: In this case, the statement is first prepared without knowing the values of the parameters, and it is then executed once the parameters are set to their values. A query with parameters is always executed using deferred execution, even if it is executed only once.

The Statement is to be Executed Several Times

Deferred execution is done using the *parse* step and the *execute* step. The *parse* step is done only once. It sends the query to the RDBMS, which prepares the execution plan. Then the *execute* step can be done several times. The same execution plan will be reused each time by the RDBMS.

The Statement Contains Parameters

In this case, placeholders for parameters must be considered. Most RDBMSs support the ISO SQL standard syntax for placeholders: the **?** symbol. Exceptions to this are Oracle® and SqlBase:

- For Oracle, the syntax can be `:<n>`, where `<n>` is an integer starting from 1.
- Oracle also uses *named parameters*. The syntax of named parameters is `:<name>`.

Example of Deferred Execution

An example of deferred execution is now presented.

First, the RDBMS connection is verified in order to use the proper placeholder syntax. This is done by checking the return value from `IldDbms::getName`. This method returns the RDBMS to which the program is connected.

```
const char* insertStr = 0 ;
if (!strncmp(dbms->getName(), "oracle", 6))
  insertStr = "insert into OPTIMS1 values (:1, :2)" ;
else
  insertStr = "insert into OPTIMS1 values (?, ?)" ;
```

This statement is parsed with the `IldRequest::parse` method:

```
if (!request->parse(insertStr)) {
  IldDisplayError("Parse of query failed : ", request) ;
  Ending(dbms) ;
  exit(1) ;
}
```

Then the parameters are bound to set their types. This parameter binding may also be used to specify other parameter information (See ***Step 3: Accessing Data Directly***, which deals with *external binding*, for further information).

```
if (!request->bindParam((IldUShort)0, IldIntegerType)) {
  IldDisplayError("First parameter binding failed : ", request) ;
  Ending(dbms) ;
  exit(1) ;
}
if (!request->bindParam((IldUShort)0, IldStringType)) {
```

```
  IldDisplayError("Second parameter binding failed : ", request) ;
  Ending(dbms) ;
  exit(1) ;
}
```

The last step is execution. A loop is run to set the parameters to different values, and the prepared query is executed with these parameter values:

```
static const IldUShort strLen = 20 ;
IldUShort i, j ;
IldInt nbRows, nVal ;
// strBuf will be used to build a different string for each execution.
char strBuf[strLen + 1] ;
strBuf[strLen] = 0 ;
for (i = 0 ; i < 5 ; i++) {
  nVal = i ;
  // Build a new string value for this execution.
  for (j = 0 ; j < strLen ; ++j)
    strBuf[j] = 'a' + i ;
  // Set parameter values.
  if (!request->setParamValue(nVal, 0) ||
      !request->setParamValue(strBuf, 1)) {
    IldDisplayError("Set parameter value failed :", request) ;
    Ending(dbms) ;
    exit(1) ;
  }
  // Execute the query.
  if (!request->execute(&nbRows, 1)) {
    IldDisplayError("Insertion failed : ", request) ;
    Ending(dbms) ;
    exit(1) ;
  }
  else cout << "\t" << nbRows << " row inserted." << endl ;
}
```

Conclusion

This step described how to run a query with parameters using the deferred execution method.

The next steps describe how to make this application even more efficient.

See source code.

Step 2: Optimizing Network Use

The goal of network optimization techniques is to reduce network traffic for a given SQL execution. Changes with respect to the standard methods (as described in ***IBM ILOG DB Link Basic Use***) are needed on two different occasions, when:

- *A Query is Sent to the RDBMS*
- *A Results Set is Retrieved for the RDBMS*

Deferred execution can also be used to execute the same query several times in only one `execute` call. This reduces the number of queries sent to the server.

Similarly, several rows can be requested at a time when retrieving results from the RDBMS. This also reduces network use. These techniques use the notion of an *Array of parameters* and an *Array of columns*.

A Query is Sent to the RDBMS

First, look at the input side, that is, when queries are sent to the RDBMS.

The same insertion as in previous steps is run, but in only one execution. To do so, you need to request an array of `n` parameters from DB Link. Then, values are set for each set of parameters. Finally, an `execute` statement is run for the entire set of parameters.

The `IldRequest::setParamArraySize` method is used to specify the size of the parameter array:

```
static const IldUShort nbParam = 5 ;
if (!request->setParamArraySize(nbParam)) {
  IldDisplayError("Could not set parameter array size : ", request) ;
  Ending(dbms) ;
  exit(1) ;
}
```

The parse and parameter binding steps are done as described in Step 1.

The parameter values are now set. This is done as in Step 1, except that the third argument of `IldRequest::setParamValue` is used to specify which parameter to set:

```
static const IldUShort strLen = 20 ;
IldInt i, j, nbRows ;
char strBuf[strLen + 1] ;
for (i = 0 ; i < nbParam ; i++) {
  nVal = i ;
  // Build a new string value for this set of parameters.
  for (j = 0 ; j < strLen ; ++j)
    strBuf[j] = 'a' + i ;
  // Set parameter values.
  if (!request->setParamValue(nVal, 0, i) ||
      !request->setParamValue(strBuf, 1, i)) {
    IldDisplayError("Set parameter value failed :", request) ;
    Ending(dbms) ;
  exit(1) ;
  }
}
```

Now, the query is run and the number of times it will be run is specified.

When deferred execution is used, by default the parameter array size is used to specify the number of times the query is to be executed. Consequently, the second argument of `IldRequest::execute(IldInt*, IldInt)` is not required (the default value is used).

The first argument of the `execute` method is set to the number of rows updated by the query (5 in this case).

```
if (!request->execute(&nbRows, nbParam)) {
IldDisplayError("Insertion failed : ", request) ;
Ending(dbms) ;
exit(1) ;
}
else cout << "\t" << nbRows << " row inserted." << endl ;
```

A Results Set is Retrieved for the RDBMS

Optimization by deferred execution can also be used when retrieving a results set from the RDBMS.

The `IldRequest::setColArraySize` method is used to specify the number of rows to retrieve in one fetch. After calling this method, the other steps of the application will be exactly the same as they would be without this optimization.

The bigger the array size, the less the server has to be contacted, and the fewer network resources are used. However, more memory is needed in this case. DB Link fetches all the rows in memory. Then, from the application point of view, the process is the same as it would be to get only one row.

The `IldRequest::fetch` method checks to see whether there is a row available in memory. If there is, the row is made available to the application. Otherwise, the method automatically gets the next block of rows from the RDBMS.

The `runDisplay` method can be seen in the complete source code for this step. This method is the same as in Step 1 (except the call to `IldRequest::setColArraySize()`).

Conclusion

This step demonstrated how several operations can be executed in only one step to reduce network use.

See source code.

Step 3: Accessing Data Directly

This step shows how to use *external binding*, as well as the advantage of using it. The external binding feature (as opposed to *internal binding*) is first presented.

Binding is the process of sending and receiving RDBMS data directly to and from the application memory.

In previous steps, the DB Link default mode was used. This mode is called internal binding. With internal binding, DB Link automatically allocates the memory where data is stored. Then, the data must be copied from the application memory to the memory allocated by DB Link. For example, in Step 1, the `IldRequest::setParamValue` method is used to copy the value needed to the buffer allocated by DB Link.

This is the easiest method since the application does not need to do an explicit binding—DB Link does it by itself. However, this method is less efficient since the value has to be copied from the application memory area to the area allocated by DB Link.

With external binding, the application allocates the memory and tells DB Link to use this area directly. This is done using one of the binding methods, `bindCol` and `bindParam`. The `bindCol` method is used when retrieving column data from the RDBMS. The `bindParam` method is used to bind application memory to the array of value arguments for a query with parameters.

To see how this works from the input side (writing data to the RDBMS), the memory to be used is first allocated. An array of parameters is used, as described in the previous step, to run only one `execute` call. This is done as follows:

```
static const IldUShort nbParam = 5 ;
// strBuf will be used to store an array of 5 string values.
char strBuf[nbParam][strLen + 1] ;
IldInt intBuf[nbParam] ;
short strNulls[nbParam], intNulls[nbParam] ;
```

This declares an array of five `strBuf` strings where string parameter values are stored.

Then, the `intBuf` array records the integer parameter values.

The `strBuf` array is used with `strNulls`, and the `intBuf` array is used with `intNulls`. These arrays record the `NULL` indicators. They are initialized with zeros to specify that the parameters are not `NULL`.

A null indicator is required to specify that the value is `NULL`. In the context of an RDBMS, a null value means that there is no value at all. For instance, for an integer column, a null value is `NULL`, not zero.

> ***Note:*** *The null indicators are not required when you do not have to handle a* `null` *value.*

Then, the parameter array size is set as described in the previous step, and the query is parsed.

The `intBuf` and `strBuf` buffers are given as arguments to the `IldRequest::bindParam` method. The use of external binding requires nothing more.

```
if (!request->bindParam((IldUShort)0,
                        IldIntegerType,
                        sizeof(IldInt),
                        intBuf,
                        intNulls)) {
  IldDisplayError("Bind first parameter failed : ", request) ;
  Ending(dbms) ;
  exit(1) ;
}
if (!request->bindParam((IldUShort)1,
                         IldStringType,
                         strLen + 1,
```

```
                          strBuf,
                          strNulls)) {
  IldDisplayError("Bind second parameter failed : ", request) ;
  Ending(dbms) ;
  exit(1) ;
}
```

Now you work directly with the buffers. They are used to send data to the RDBMS.

```
IldUShort i, j ;
IldInt nbRows ;
for (i = 0 ; i < nbParam ; i++) {
  intBuf[i] = i ;
  // Build a new string value for this set of parameters.
  for (j = 0 ; j < strLen ; ++j)
    strBuf[i][j] = 'a' + i ;
}
if (!request->execute(&nbRows, nbParam)) {
  IldDisplayError("Insertion failed : ", request) ;
  Ending(dbms) ;
  exit(1) ;
}
else cout << "\t" << nbRows << " row inserted." << endl ;
```

Conclusion

This step demonstrates how to bind DB Link directly to application memory. This is an efficient way of exchanging data with the RDBMS.

See source code.

Step 4: Keeping Data in Memory

This step shows how to use multiple binding, as well as the advantage of using it. Multiple binding is used to get better performance when the application needs to keep in memory the objects it reads from the RDBMS.

In previous steps, data always went to the same address location when it was retrieved from the RDBMS. Therefore, each fetch operation overwrites the data previously fetched. If the application needs to keep this data in memory, it has to copy it to another location (this can lead to memory exhaustion problems).

To avoid this, the `IldRequest::bindCol` method is called as often as necessary to specify a new address location between each fetch. This is demonstrated by the following sample (see method `readData` in the complete code sample for this step).

The *select* query is executed as in previous steps:

```
static const char* selectStr = "select I from OPTIMS4" ;
if (!request->execute(selectStr)) {
  IldDisplayError("Could not run select query : ", request) ;
  Ending(dbms) ;
  exit(1) ;
}
```

A first `bindCol` call is executed to get the first column value. Then, within the fetch loop, the column binding is changed so that each value fetched is stored in a new location:

```
IldUShort i = 0 ;
if (!request->bindCol((IldUShort)0, IldIntegerType, &values[i])) {
  IldDisplayError("Could not bind column : ", request) ;
  Ending(dbms) ;
  exit(1) ;
}
while (request->fetch().hasTuple())
  if (++i == nbVal) break ;
  else
    if (!request->bindCol((IldUShort)0, IldIntegerType, &values[i])) {
      IldDisplayError("Could not bind column : ", request) ;
      Ending(dbms) ;
      exit(1) ;
    }
```

> ***Note:*** *In this sample, only the first values in an array are retrieved. This is why the fetch loop is broken after a given number of rows. In a real application, memory used to store data is allocated dynamically as required. Data is then printed at the end of the program.*

Conclusion

This step demonstrated how to use the multiple binding method to keep in memory data read from the RDBMS. This is easy to use and avoids the possible memory exhaustion problems that can arise when copying the data to another application buffer.

See source code.

IBM ILOG DB Link Portability Considerations

A major feature of DB Link is its portability across various RDBMSs and systems. An application built for Oracle®, for instance, will also work with Informix®, on either Solaris™ or Windows®.

This chapter describes specific considerations to keep in mind when building portable applications:

- ***Step 1: Using the Date as Object Mode***,
- ***Step 2: Using the Numeric as Object Mode***,
- Avoiding problems that arise from various LOCALE settings. This is discussed in both steps.

Step 1: Using the Date as Object Mode

In its default configuration, DB Link handles the date as a string. This is referred to as the *date as object* mode. In this mode, date strings must respect the format expected by the RDBMS. This format varies depending on the RDBMS being connecting to. Also, this format depends on the LOCALE setting.

> ***Note:*** *Some RDBMSs can handle time with milliseconds. The "date as string" mode does not allow you to get these milliseconds, whereas "date as object" does. Consequently, the "date as object" mode respects the precision of the data returned by the RDBMS.*

To avoid these dependencies, DB Link provides a class (`IldDateTime`) to record a date. This object may be used to send or retrieve a date to or from the RDBMS. DB Link silently converts this to what is expected by the RDBMS (a specific structure).

This `IldDateTime` class provides logical accessors to build the date value. The sample `PortStep1.cpp` shows how to use this class to record date values to a table.

Since the default mode for dates is *date as string*, you first switch to date as object mode. To do so, the `IldIldBase::setStringDateUse(IldBoolean)` method is used:

```
request->setStringDateUse(IldFalse) ;
```

`IldDateTime` objects can now be used.

An insert query is run to insert new date values in a table. An array of parameters is used, with external binding, as described in *IBM ILOG DB Link Optimization Techniques*.

```
if (!request->parse(insertStr))
  IldDisplayError("Could not parse insert query", request) ;

if (!request->bindParam((IldUShort)0, IldDateTimeType, sizeof(IldDateTime),
                        dates, dateNulls))
  IldDisplayError("Bind parameter failed : ", request) ;
```

The values for the date parameters are now set. This is done using the intuitive interface provided by the `IldDateTime` class.

```
for (i = 0 ; i < nbParam ; ++i) {
  dates[i].setYear(1999) ;
  dates[i].setMonth(10) ;
  dates[i].setDay(i + 1) ;
  dates[i].setHour(10) ;
  dates[i].setMinute(30) ;
}
```

The query is now executed:

```
if (!request->execute(&rowCount, nbParam)) {
  IldDisplayError("Could not execute insert query : ", request) ;
  Ending(dbms) ;
  exit(1) ;
```

```
}
```

Then the data is retrieved using the method `displayData`. This is done in string mode, since the values need only to be printed. Using this sample, you can check how the date strings will be affected by the RDBMS format and the LOCALE settings.

Conclusion

This step demonstrated the use of the `IldDateTime` class. This class provides an intuitive way to handle dates and exchange date data with the RDBMS, regardless of LOCALE settings. Without DB Link, this requires knowledge of the specific structures used by the RDBMS API. Such code is complicated and is not portable to other RDBMSs.

See source code.

Step 2: Using the Numeric as Object Mode

In this step you will see the following items:

- *The Numeric as Object Mode*
- *Setting the Numeric Mode*
- *Displaying the Current Numeric Mode*
- *Inserting Data Using the Numeric as Object Mode*
- *Executing the Query*

The Numeric as Object Mode

The *numeric as object* mode is similar to the date as object mode. Instead of handling the numeric value as a string (which depends on the LOCALE setting for the decimal separator), or as a C float data type (which implies a loss of precision, since database numeric types can handle a precision much greater than a float), DB Link contains a C++ class called `IlNumeric`. This class provides an intuitive interface to numeric values. DB Link converts the value in an RDBMS-specific structure without any loss of precision and independently of LOCALE settings.

Two other possible modes are:

- With the default mode, numeric values are handled as basic C++ types—either integers or float values. Since large numbers cannot be represented in this mode, precision may be lost. You can return to this default mode by using either `IldIldBase::setNumericMode(IldFalse)` or `IldIldBase::setStringNumericMode(IldFalse)`.
- With the numeric as string mode, numeric values are handled as strings. While there is no loss of precision, the string representation will depend on the LOCALE settings. The `IldIldBase::setStringNumericUse(IldTrue)` method activates this mode.

The best way to handle large floating values is to use the numeric as object mode, which does not lose precision and is independent of the LOCALE settings. This mode is activated with method `IldIldBase::setNumericUse(IldTrue)`.

Setting the Numeric Mode

The following code demonstrates how to set the numeric mode. It also demonstrates the effect of this mode on the results set retrieved from the database. In method `checkNumericMode`, a new request is opened and then successively set to the three different modes. For each mode, a description of the mode is printed. Here is an excerpt from this method:

```
request->setStringNumericUse(IldTrue) ;
displayNumericMode(dbms, request, "Mode \"Numeric as String\" is activated :",
                   IldStringType) ;

request->setNumericUse(IldTrue) ;
displayNumericMode(dbms, request, "Mode \"Numeric as Object\" is activated :",
                   IldNumericType) ;
```

Displaying the Current Numeric Mode

The `displayNumericMode` method displays the current numeric mode using the methods `IldIldBase::useNumeric` and `IldIldBase::useStringNumeric`. Then, it runs a *select* query to select a numeric value from the database. Depending on the current numeric mode, this column will be of type `IldRealType` (default mode), `IldStringType` (numeric as string mode), or `IldNumericType` (numeric as object mode).

Here is the code of the `displayNumericMode` method:

```
static const char* selectStr = "select N from PORTS2" ;
cout << message << endl ;
     << "* request->useNumeric() = "
     <<(request->useNumeric() ? "IldTrue" : "IldFalse")
     <<", request->useStringNumeric() = "
     (request->useStringNumeric() ? "IldTrue" : "IldFalse") << endl ;
if (!request->execute(selectStr)) {
  IldDisplayError("Select execution failed : ", request) ;
  Ending(dbms) ;
  exit(1) ;
}
```

Inserting Data Using the Numeric as Object Mode

Data is now inserted in the table using the numeric as object mode. This is done with method `insertData`. First, numeric as object mode is activated:

```
request->setNumericUse(IldTrue) ;
```

The insert query is prepared as in previous steps. To bind the parameter, `IldNumericType` is used to specify the parameter in numeric as object mode. An array of two parameters is used:

```
if (!request->parse(insertStr)) {
```

```
  IldDisplayError("Could not parse insert query : ", request) ;
  Ending(dbms) ;
  exit(1) ;
}
if (!request->bindParam((IldUShort)0, IldNumericType, sizeof(IlNumeric), nums,
     numNulls)) {
  IldDisplayError("Bind parameter failed : ", request) ;
  Ending(dbms) ;
  exit(1) ;
}
```

Executing the Query

The two numeric objects are initialized with a string of characters. They can also be initialized with a double-precision value, but this is not as precise. The query is then executed:

```
// Set the values for the numbers :
nums[0].set("1234567890.456") ;
nums[1].set("-86420.13579") ;

// Initialize null buffers :
memset(numNulls, 0, sizeof(short) * nbParam) ;

if (!request->execute(&rowCount, nbParam)) {
  IldDisplayError("Could not execute insert query : ", request) ;
  Ending(dbms) ;
  exit(1) ;
}
```

Conclusion

This step demonstrated the use of the `IlNumeric` class. This class provides an intuitive way to handle large floating values with the RDBMS, regardless of LOCALE settings and with no loss of precision. Without DB Link, this requires knowledge of the specific structures used by the RDBMS API. Such code is complicated and is not portable to other RDBMSs. This step also demonstrated how the three different numeric modes are activated.

See source code.

IBM ILOG DB Link Access to Object Data Types

This chapter demonstrates how IBM® ILOG® DB Link gives access to the new data types introduced by an object oriented RDBMS. DB Link supports Informix Universal Server and Oracle®, which both provide object oriented features. In this document, these new data types are called *Abstract Data Types* (ADT).

There are 2 basic kinds of structures introduced by these new RDBMSs—**lists**, and **objects**.

First, this chapter demonstrates how to describe such a data type. It presents all the methods available to access the structure of the data type. To work with the objects created in the

database, you need to know how these objects are built, that is, what their attributes are. This is referred to as describing the Abstract Data Type.

Then, the chapter presents the classes used to build an instance of an Abstract Data Type within DB Link, and how this instance is stored in the RDBMS.

These 2 items are reflected in this document as follows:

- ***Step 1: Getting the Description of an ADT*** - Shows how to get the description of an ADT.
- ***Step 2: Building an ADT*** - Shows how to handle an ADT value.

Step 1: Getting the Description of an ADT

This step demonstrates how to get the description of an ADT. An `IldADTDescriptor` instance is retrieved. This gives us the description of the ADT, since an ADT is described using DB Link class `IldADTDescriptor`.

In this step, the following items are presented:

- ***Objects and Abstract Data Types***
- ***Creating an ADT Instance***
- ***Displaying the Object Structure***
- ***Printing the ADT Attributes***

Objects and Abstract Data Types

There are various kinds of object types in an RDBMS. However, these object types can be different with Informix and Oracle®. DB Link provides two basic object types: `IldObjectType` and `IldCollectionType`. When an ADT column is retrieved, it will be one of these 2 types. `IldObjectType` may be the *object* type in Oracle, and *named row* or *unnamed row* in Informix. `IldCollectionType` may be *varray* or *nested table* types in Oracle, or *nested table*, *list*, *set*, or *multiset* in Informix Universal Server.

The `IldADTDescriptor` class provides an additional type name to give more information on the exact data type in the RDBMS. This is the `IldADTType` returned by `IldADTDescriptor::getType`. Depending on this type, you may want to access a specific ADT attribute. For instance, a varray is a specific kind of collection, since there is a limit to the maximum number of elements that can be recorded in the collection. This maximum number of elements can be retrieved from an `IldADTDescriptor` instance and is meaningful only for a varray ADT.

The following table summarizes the various ADT types handled by DB Link.

DB Link Column type 'IldColumnType'	IldADTDescriptor type 'IldADTType'	Database type
`IldObjectType`	`IldADTObject`	Oracle® objects, and Informix [named] rows
`IldCollectionType`	`IldADTTable`	Informix or Oracle nested tables
`IldCollectionType`	`IldADTList`	Informix lists, sets, and multisets
`IldCollectionType`	`IldADTArray`	Oracle varrays

In DB Link, the various list types (`IldADTTable`, `IldADTList`, and `IldADTArray`) are manipulated in the same way. However, specific information that depends on the `IldADTType` can be retrieved.

To access an object type, you first create one within the database. This is done by the `createADT(IldDbms*)` method in source file `ADTCommon.cpp`.

> ***Note:*** *The SQL commands used to build the object data type depend on the database used—Oracle® or Informix US.*

The object structure is made from one or more of the following object data types:

- POINT: A POINT object contains the coordinates of the point—two integer values X and Y.
- LINE: A LINE object contains two POINT objects.
- BRIDGE: A BRIDGE object contains a name and a nested object LINE.
- BRIDGELST: A BRIDGELST object data type contains a list of bridges. With Informix, a data type name cannot be given to a collection. Therefore, the BRIDGELST data type cannot be created in this step. It will be created in Step 2, within a table.

All this makes for a complex nested object structure. This structure is described with DB Link.

Creating an ADT Instance

The program code first creates an `IldADTDescriptor` instance for the data type to be described. To do so, the name of the object data type is specified.

```
IldADTDescriptor* adt = 0 ;
if (!(adt = dbms->readAbstractType(checkCase(ADTName, dbms)))) {
  IldDisplayError("Could not getADT description : ", dbms) ;
  Ending(dbms) ;
```

```
  exit(-1) ;
}
```

> ***Note:*** *In this call, the* `checkCase` *method is used. This ensures that the name of the data type is spelled in the correct case, depending on the RDBMS. With Informix, it is of the BRIDGE data type. With Oracle®, it is of the BRIDGELST data type. In this code excerpt, 'ADTName' contains this name—either BRIDGE for Informix or BRIDGELST for Oracle.*

Another way to get an `IldADTDescriptor` instance is to use the data type ID instead of its NAME. This is mainly useful for Informix data types that are not named: unnamed rows and the various kinds of collections. This method is not used here.

Displaying the Object Structure

Now the structure of the object is displayed. This is done recursively, since the object contains nested objects.

First, the type of the object is tested using method `IldADTDescriptor::getType`.

- If the `IldADTDescriptor` object describes an object (`IldObjectType`), the following methods are used:
  - `IldADTDescriptor::getAttributes` - Returns an array of `IldDescriptor` objects. Each attribute of the ADT is described by one of these `IldDescriptor` objects.
  - `IldADTDescriptor::getAttributesCount` - Gives the number of attributes of the ADT. This is also the number of elements in the array returned by `IldADTDescriptor::getAttributes`.
- When the object is a collection, it is built upon only one attribute. Hence, you do not have access to all the information (for example, the maximum number of elements in the list). To retrieve this additional information, the following methods are used:
  - `IldADTDescriptor::getCollectionAttribute` - This is equivalent to `IldADTDescriptor::getAttributes`, but it returns only one element.
  - `IldADTDescriptor::getCollMaxSize` - Returns the maximum number of elements in the list when the collection type is `IldADTArray`.

The code to get the description of the ADT is as follows (method `displayADT`):

```
IldUShort i = 0 ;
switch (adt->getType()) {
case IldADTObject : {
  const IldDescriptor* const* elts = 0 ;
  cout << "Object (" ;
  elts = adt->getAttributes() ;
  for (i = 0 ; i < adt->getAttributesCount() ; i++) {
    displayDesc(dbms, elts[i]) ;
```

```
    if (i < adt->getAttributesCount() - 1)
      cout << ", " ;
  }
  cout << ")" ;
  break ;
}
case IldADTList :
case IldADTArray : {
  const IldDescriptor* desc = adt->getCollectionAttribute() ;
  if (adt->getType() == IldADTList)
    cout << "List of {" ;
  else
    cout << "List[" << adt->getCollMaxSize() << "] of {" ;
  displayDesc(dbms, desc) ;
  cout << "}" ;
  break ;
}
default:
  cout << "Unexpected ADT Type." << endl ;
}
```

Here, the `IldDescriptor` instances that describe each attribute of the ADT are retrieved from the `IldADTDescriptor` instance that describes the type. Then, the description of each of these attribute descriptions is printed.

Printing the ADT Attributes

Printing the ADT attributes is done by the method `displayDesc`. If one of the attributes is an object (nested object), the `displayDesc` method recursively calls `displayADT` to print the ADT description.

Here is an excerpt from method `displayDesc`:

```
cout << desc->getName() ;
switch (desc->getType()) {
case IldObjectType :
case IldCollectionType :
  if (desc->getADTDescriptor()->isNamedType())
    cout << "'" << desc->getSqlTypeName() << "' : " ;
  displayADT(dbms, desc->getADTDescriptor()) ;
  break ;
case IldStringType :
  cout << desc->getSqlTypeName() << " (" << desc->getSize() << ")" ;
  break ;
case IldIntegerType :
  cout << desc->getSqlTypeName() ;
  break ;
default:
  cout << "Other type : " << desc->getType() << endl ;
}
```

Conclusion

This step demonstrated how to get the description of an Abstract Data Type. The two methods `displayADT` and `displayDesc` show how to go through the `IldADTDescriptor` instance to get the description of the various attributes of the data type.

See `ADTCommom.cpp` source code (creation of the object types).

See main source code.

Step 2: Building an ADT

This step demonstrates how to build a value for an Abstract Data Type and send it to the RDBMS. (The same database objects as in the previous step is used).

Building the ADT is done as follows:

- ***Creating the Table***
- ***Getting the ADT Description***
- ***Retrieving the Instances***
- ***Building the Object Values***
- ***Executing the Query***

Creating the Table

First, the table ADTS2 is created. The table contains 3 fields:

- The name of a river (RIVER)
- The length of the river (LENGTH)
- A list of bridges associated with the river (B)

Then an ADT value is built and recorded in the B field.

Getting the ADT Description

The ADT value is an instance of the DB Link class `IldADTValue`. To build such an instance, the description of the Abstract Data Type is needed.

Since you now have a table that uses the ADT, another way of getting this description is presented (different from the method used in the first step). All that is required is to run a query to select the *B* column from the *ADTS2* table. Then, the description of the column can be accessed, which means that the description of its data type can also be accessed.

This is done in the following code excerpt:

```
// Retrieve the descriptor of the parameter object type.
IldADTDescriptor* bridgeLstAdt = 0 ;
const char* query = "select B from ADTS2" ;
if (!request->execute(query)) {
  IldDisplayError("Could not select object column : ", request) ;
  localEnd(dbms) ;
  exit(1) ;
}
bridgeLstAdt = request->getColDescriptor(0)->getADTDescriptor() ;
```

You now have the `IldADTDescriptor` instance that describes the upper level object: the list of bridges. This instance is required to build the ADT value and also to bind the Abstract Data Type parameter. The query parse operation and the binding of the parameter is similar to what has been done in previous steps with basic types. The only difference is that for an ADT parameter, the `IldADTDescriptor` of the data type must be provided. This is done as follows:

```
// Parse the insert query :
const char* insertStr =
       (!strncmp(dbms->getName(), "oracle", 6) ?
       "insert into ADTS2 values ('River Name', 30, :1)" :
       "insert into ADTS2 values ('River Name', 30, ?)") ;

cout << "Parse request : " << insertStr << endl ;
if (!request->parse(insertStr)) {
  IldDisplayError("Could not parse insert query : ", request) ;
  localEnd(dbms) ;
  exit(1) ;
}

if (!request->bindParam((IldUShort)0, IldCollectionType, -1, 0, 0, IldFalse, 0,
    bridgeLstAdt)) {
  IldDisplayError("Could not bind object parameter : ", request) ;
  localEnd(dbms) ;
  exit(1) ;
}
```

Retrieving the Instances

The request is now parsed and a value given to the parameter. Since the main object (the list of bridges) contains inner objects, you also need to get the `IldADTDescriptor` instances that describe these nested objects. These instances are retrieved through the upper level `IldADTDescriptor`.

This is done in the method `getSubADTDescriptor`, which requires the following parameters:

- `const IldADTDescriptor* adt` - The main object.
- `IldUShort idx` - The index of the nested object to access.

> ***Note:*** *The method has two other parameters. These are used only to process the error cases.*

The inner object `IldADTDescriptor` is retrieved as follows:

```
IldADTDescriptor* subAdt = 0 ;
// Get the descriptor at the given position :
if (adt->getType() == IldADTObject) {
  if (adt->getAttributesCount() > idx)
    desc = adt->getAttributes()[idx] ;
}
else
  desc = adt->getCollectionAttribute() ;
```

```
// Get the ADT descriptor :
subAdt = desc->getADTDescriptor() ;
```

Then, the `getSubADTDescriptor` method is used to get each ADT descriptor for each object nested within the bridge collection:

```
// Get the ADT Descriptor for the bridge object :
IldADTDescriptor* bridgeAdt = getSubADTDescriptor(dbms, bridgeLstAdt, 0,
                                                  "bridge") ;
// Get the ADT Descriptor for the line object :
IldADTDescriptor* lineAdt = getSubADTDescriptor(dbms, bridgeAdt, 0, "line") ;
// Get the ADT Descriptor for the point object :
IldADTDescriptor* pointAdt = getSubADTDescriptor(dbms, lineAdt, 0, "point") ;
```

Building the Object Values

You now have everything required to build the object values. This is done from the most nested level to the upper level as follows:

```
IldADTValue* pointObj1 = new IldADTValue(pointAdt) ;
IldADTValue* pointObj2 = new IldADTValue(pointAdt) ;
IldADTValue* lineObj1 = new IldADTValue(lineAdt) ;
IldADTValue* bridgeObj1 = new IldADTValue(bridgeAdt) ;
IldADTValue* bridgeLst = new IldADTValue(bridgeLstAdt) ;

pointObj1->setValue((IldInt)10, 0) ; // X for point 1.
pointObj1->setValue((IldInt)20, 1) ; // Y for point 1.
pointObj2->setValue((IldInt)10, 0) ; // X for point 2.
pointObj2->setValue((IldInt)30, 1) ; // Y for point 2.

lineObj1->setValue(pointObj1, 0) ; // First point of the line.
lineObj1->setValue(pointObj2, 1) ; // Second point of the line.
bridgeObj1->setValue(lineObj1, 0) ;
bridgeObj1->setValue("Bridge Name", 1) ;

bridgeLst->setValue(bridgeObj1, 0) ;
```

Executing the Query

The parameter value is then set and the query executed as done with basic data types in previous steps:

```
if (!request->setParamValue(bridgeLst, 0)) {
  IldDisplayError("Could not set parameter value : ", request) ;
  localEnd(dbms) ;
  exit(1) ;
}

if (!request->execute(&rowCount, 1)) {
  IldDisplayError("Could not execute the query : ", request) ;
  localEnd(dbms) ;
  exit(1) ;
}
else
  cout << rowCount << " rows inserted." << endl ;
```

The same method is used to add a second row with a list that contains three elements.

You can also look at the method `displayData`, which retrieves from the database the objects previously recorded.

Conclusion

This step demonstrated how to build a value for an Abstract Data Type and how to record it in the database. It is much simpler to do so with DB Link than with the native RDBMS API.

See `ADTCommom.cpp` source code (creation of the object types).

See main source code.

Index

R

T