IBM

Accessing IBM DB2 for i data from Linux applications running on IBM i

Optimize Linux access to your DB2 for i database

.

Kent Milligan
Systems and Technology Group ISV Enablement
March 2014

@IBMSystemsISVs

Table of contents

Abstract

This paper provides implementation tips and a number of coding examples that illustrate how to optimize the access to your IBM DB2 for i database from Linux systems using middleware provided by IBM for ODBC and JDBC access.

Introduction

The Linux® operating system, an open-source implementation of the UNIX® operating system, is rapidly gaining acceptance among users of the IBM® Power Systems™ family of servers. With IBM Power Systems, partitions can run Linux alongside partitions running the IBM i and IBM AIX® operating systems. Traditionally, many organizations use the IBM i platform as a reliable and highly scalable database server. The IBM i database, IBM DB2® for i, is fully integrated into the IBM i operating environment and uses its robust features, such as single-level storage, tight security, and object-based architecture. For these reasons, IBM believes that DB2 for i remains the database of choice for most IBM i shops.

Generally, Linux applications use ODBC or JDBC connections to access business data stored in DB2 for i databases. Applications can use either the database middleware offerings that are included with the IBM i operating system or those provided by the IBM DB2 Connect™ product. This paper details how to use these middleware solutions and the differences between them.

Note that the Linux command and utility examples included in this paper were run using the Red Hat Enterprise Linux 6 distribution. The command syntax and location of directories might differ slightly for other Linux distributions.

Using the IBM i middleware

The IBM i operating system includes database middleware to make access of DB2 for i databases simple and easy for application developers. ODBC drivers are available with the IBM i Access software brand while the IBM Toolbox for Java provides a JDBC driver. As you might expect, these drivers only support access of DB2 for i databases due to the tight integration with the IBM i operating system.

Using the IBM i Access ODBC driver

IBM i operating system actually includes the following two ODBC drivers for Linux clients that are provided at no additional charge:

- IBM i Access ODBC driver from the IBM i Access Client Solutions Linux Application Package (refer: **ibm.com**/systems/power/software/i/access/solutions.html)
- IBM iSeries® Access for Linux ODBC Driver

Both of these drivers are based on the ODBC driver that is provided by IBM i Access for Windows (formerly known as System i Access for Windows and iSeries Access for Windows). These Linux ODBC drivers use the IBM i database host servers as the access point to the system and communicate with the back-end database server job through a socket connection. Both Linux ODBC drivers also provide support for Linux running on IBM PowerPC® and Intel® Xeon® processors.

IBM recommends the usage of the IBM i Access Linux ODBC driver from the IBM i Access Client Solutions Linux package because this is the only Linux ODBC driver that IBM plans on enhancing. The IBM i Access ODBC driver provides the following key advantages when compared with the iSeries Access for Linux ODBC driver.

- Provides support for latest versions of unixODBC on 64-bit platforms. ODBC applications and abstraction layers such as PHP ODBC compiled on recent 64-bit Linux systems do not work with the iSeries Access for Linux ODBC driver
- Provides support for PowerPC version of unixODBC (requires December 2013 service pack)
- Provides support for both RPM and deb packages. This should help those of you who are using Debian or Ubuntu to access your IBM i
- Provides support for 32-bit co-installable libraries by including additional RPM and deb packages to install the 32-bit libraries along with the 64-bit package
- Includes recent fixes and enhancements from the IBM i Access for Windows ODBC driver

Getting the IBM i Access for Linux ODBC driver

The IBM i Access Client Solutions Linux Application Package can be downloaded from IBM Entitled Software Support (ESS) (at **ibm.com**/servers/eserver/ess/index.wss) by any customer with an IBM i software maintenance contract. For information on how to easily download the packages, refer to Obtaining Access Client Solutions (at **ibm.com**/support/docview.wss?uid=nas8N1010355).

Installation

After downloading the package to your system, you need to extract the contents of the .zip file and then find the directory appropriate for your system's architecture. Developers wanting to use the ODBC driver from a Linux partition on a Power System server would use the installation image in the ppc_64Bit directory. The directories contain both .rpm and .deb installers

To install this Linux application package, you can use the package manager suitable for your Linux distribution; this includes zipper, yum, apt-get, rpm, or dpkg. Here is an example installation request that uses the `rpm` command.

```
rpm –i ibmi-access-1.1.0.1-1.0.ppc64.rpm
```

Unfortunately, when the unixODBC project added full 64-bit support in unixODBC 2.2.14, the shared library version did not get incremented from s*o.1* to *so.2*. This library version was finally fixed with unixODBC 2.3.1, but many distributions are not yet including this version. Many distributions did realize that there was a break in compatibility and updated their packages to reflect this. On Red Hat, Fedora, and others, you might see an error when trying to install the IBM i Access Client Solutions Linux Application Package about a missing dependency on libodbcinst.so.1. To fix this, you need to force install the package and create a symbolic link from libodbcinst.so.2 to libodbcinst.so.1.

Here is an example invocation of the `rpm` command to force the installation:

```
rpm –i --nodeps ibmi-access-1.1.0.1-1.0.ppc64.rpm
```

The following instructions specify how to create the required symbolic links for each of the distributions.

- 64-bit RPM-based distributions:
  `ln -s libodbcinst.so.2 /usr/lib64/libodbcinst.so.1`
- 32-bit RPM-based distributions:
  `ln -s libodbcinst.so.2 /usr/lib/libodbcinst.so.1`
- 64-bit Debian-based distributions:
  `ln -s libodbcinst.so.2 /usr/lib/x86_64-Linux-gnu/libodbcinst.so.1`
- 32-bit Debian-based distributions:
  `ln -s libodbcinst.so.2 /usr/lib/i386-Linux-gnu/libodbcinst.so.1`

IBM hopes to address this library version issue in a future release.

You can quickly verify that the driver works properly by using the CWBPING utility that is part of the driver installation package. The mysystem identifier in this example invocation of the utility is the TCP/IP host name of the IBM i system that is being accessed:

```
/opt/ibm/iSeriesAccess/bin/cwbping mysystem
```

Configuration

The driver installation process registers the IBM i Access Linux ODBC driver with the unixODBC driver manager by adding the driver information to the odbcinst.ini configuration file. For this installation, this file was located in the /etc directory. The `odbc_config` utility can be used to locate this configuration file with the following invocation: `odbc_config --odbcinstini`

Here is an example of the content within the odbcinst.ini file:

```
[IBM i Access ODBC Driver]
Description = IBM i Access for Linux ODBC Driver
Driver      = /opt/ibm/iSeriesAccess/lib/libcwbodbc.so
Setup       = /opt/ibm/iSeriesAccess/lib/libcwbodbcs.so
Driver64    = /opt/ibm/iSeriesAccess/lib64/libcwbodbc.so
Setup64     = /opt/ibm/iSeriesAccess/lib64/libcwbodbcs.so
Threading   = 2
DontDLClose = 1
UsageCount  = 1

[IBM i Access ODBC Driver 64-bit]
Description = IBM i Access for Linux 64-bit ODBC Driver
Driver      = /opt/ibm/iSeriesAccess/lib64/libcwbodbc.so
Setup       = /opt/ibm/iSeriesAccess/lib64/libcwbodbcs.so
Threading   = 2
DontDLClose = 1
UsageCount  = 1
```

Even though the installer updates the odbcinist.ini automatically, it is important to understand the contents of this file because the ODBC driver name (values in brackets) is required when constructing a data source name (DSN) entry for your DB2 for i database.

The next step in the configuration process is to create a DSN entry for the DB2 for i database that needs to be accessed. This can be accomplished by editing the odbc.ini configuration file.

There are two types of DSN entries: system DSN and user DSN. The system DSN is usually created by root and resides in the /etc directory. Any user on the system can use a system DSN. A user DSN is scoped to a particular user and cannot be shared with other users on the system. The configuration file for a user DSN is called .odbc.ini and is located in the user's home directory (for instance, /home/userabc/.odbc.ini).

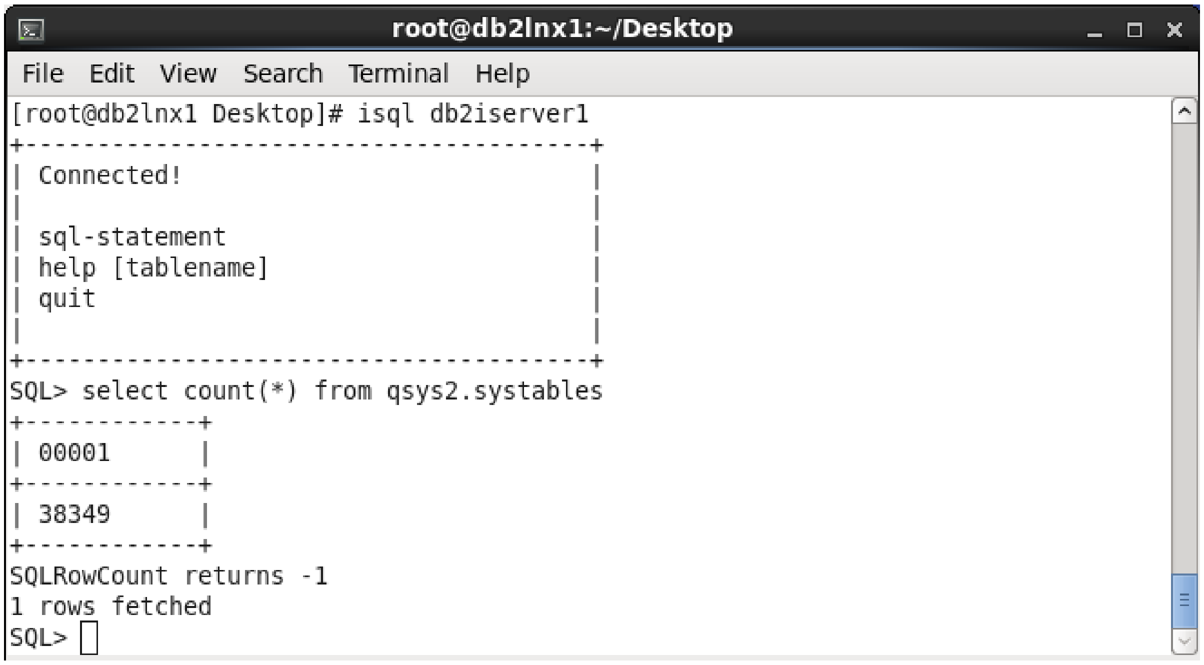
Here is an example entry for a system DSN added to the odbc.ini file:

```
[db2iserver1]
Description = IBM i Access ODBC driver
Driver = IBM i Access ODBC driver
System = mysystem
UserID = db2user
Password = userpwd
Naming = 0
```

The data source name, `db2iserver1`, is encapsulated in brackets. The DSN value is an arbitrary value that can be selected independent of your IBM i system. The `System` keyword is where you specify the TCP/IP host name of the target IBM i partition or server that needs to be accessed. In this example, that host name value is `mysystem`. The `UserID`, `Password`, and `Naming` keywords are optional. Connection keywords are covered in more detail in a later section.

If you have trouble determining the location of the odbc.ini file, the `odbc_config` utility can be invoked with the `--odbcini` option to determine which odbc.ini file is being used by the unixODBC driver manager.

After the system DSN exists, you can use the `isql` utility (part of the unixODBC Driver Manager installation) to obtain an ODBC connection to the target DB2 for i database and to run SQL statements. The utility is pretty simple, but it is useful for testing your ODBC connection and running basic SQL statements. Figure 1 shows a sample `isql` session.



```
[root@db2lnx1 Desktop]# isql db2iserver1
+---------------------------------------+
| Connected!                            |
|                                       |
| sql-statement                         |
| help [tablename]                      |
| quit                                  |
|                                       |
+---------------------------------------+
SQL> select count(*) from qsys2.systables
+------------+
| 00001      |
+------------+
| 38349      |
+------------+
SQLRowCount returns -1
1 rows fetched
SQL>
```

Figure 1: Use an isql session to run SQL statements

Changes to existing applications

Applications using the original iSeries Access Linux ODBC driver might need to be updated to work correctly with the new ODBC driver. Applications that use SQLLEN most likely just need a recompile. Microsoft® changed the SQLINTEGER parameters in many application programming interfaces (APIs) to SQLLEN (applications that intermixed the two might now get a bunch of errors about conflicting types). This requires the application to be modified to use SQLLEN instead of SQLINTEGER. For more information on changes for full 64-bit compatibility, refer to the Microsoft article, *ODBC 64-Bit Information*, at http://msdn.microsoft.com/en-us/library/windows/desktop/ms716287%28v=vs.85%29.aspx.

Applications using an abstraction layer such as PHP, Ruby or other scripting languages should not have any issues. One exception to this claim is support for PDO_ODBC which has an error (refer to https://bugs.php.net/bug.php?id=50444) that produces errors with the 64-bit version of the Linux

ODBC driver. Users can either rebuild PDO_ODBC or use the 32-bit version of the Linux ODBC driver.

Example ODBC application

This section covers the ODBC implementation details for a sample C++ program called `testodbc`. To compile the source, use the following command in the Linux partition:

```
g++ -o testodbc –lodbc testodbc.cc
```

The application retrieves a number of rows from the QCUSTCDT table in the QIWS schema (or library). The selection criterion is based on the current value of the `state` column. Here is the `SELECT` statement used by the testodbc program:

```
SELECT lstnam, city FROM qiws.qcustcdt WHERE state = ?
```

Figure 2 shows the software components involved in the running of this sample application.

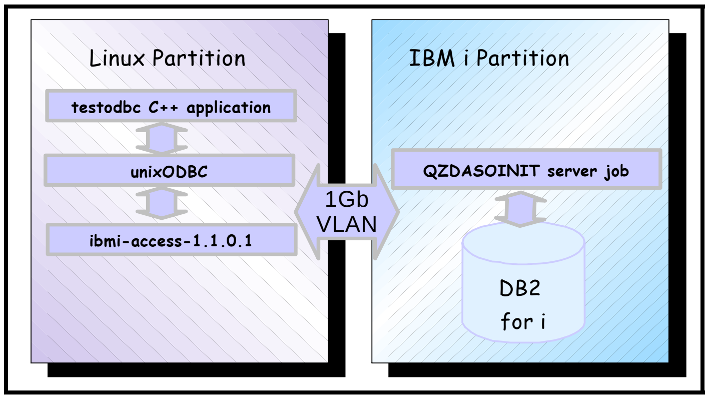


Figure 2: How an ODBC client accesses DB2 for i database

Although the testodbc application is fairly simple, it covers some important aspects of efficient ODBC programming for DB2 for i.

Addressing locale issues

Let us start by examining the class constructor of the testodbc program, as seen in Example 1.

```
TestODBC_Class::TestODBC_Class()
{
   // Set the locale for the application
   setlocale( LC_ALL, "" ); [1]

   // Allocate an environment handle
   rc = SQLAllocHandle(SQL_HANDLE_ENV, SQL_NULL_HANDLE, &EnvHandle);

   // Set the ODBC application version to 3.x
   if (rc == SQL_SUCCESS)
      rc = SQLSetEnvAttr(EnvHandle, SQL_ATTR_ODBC_VERSION,
                         (SQLPOINTER) SQL_OV_ODBC3, SQL_IS_UINTEGER);

   // Allocate a connection handle
   if (rc == SQL_SUCCESS)
```

```
      rc = SQLAllocHandle(SQL_HANDLE_DBC, EnvHandle, &ConHandle);
}
```

Example 1: Class constructor for the sample application

At **[1]**, the locale is set for the application. This is important in cases where the Linux application uses the entire character set for a given client code page. For instance, it might use a special character, such as a tilde (~), as a keyword delimiter in a literal within a WHERE clause. The IBM i ODBC driver uses the locale that is set up by the application to convert the SQL statements. If the application does not set up the locale in its class constructor or main() function, the default compiler "C" locale is used. This means that a client code page of 367 is used for some client conversions. These conversions do not handle much outside of the a-z, 0-9 character range. As a result, an SQL statement might not be properly converted to the database-server code page. In this case, DB2 for i returns an SQLCODE error value of –104 (Token was not valid). By specifying the locale in the application, this risk is eliminated.

Connection keywords

The IBM i Access for Linux ODBC driver supports a number of connection keywords that control the connection properties and improves the application's performance. The IBM i Information Center (at: http://pic.dhe.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzaik%2Fconnectkeywords.htm) has a complete list of the connection keywords.

There are several methods to specify the connection attributes. For instance, you can modify the DSN entry in the odbc.ini file. Example 2 shows an example of a system DSN entry.

```
[db2iserver1]
Description = IBM i Access ODBC driver
Driver = IBM i Access ODBC driver
System = mysystem
CommitMode = 2
ExtendedDynamic = 1
DefaultPkgLibrary = DB2USER
DefaultPackage = A/DEFAULT(IBM),2,0,1,0,512
AllowUnsupportedChar = 1
```

Example 2: An example of a system DSN entry

Alternatively, you can specify the connection attribute keywords on the connection string in the application. Example 3 shows a code snippet that illustrates this method.

```
char  ConnectStr[512] = "DSN=db2iserver1;UID=db2user;PWD=userpwd;";
…
   // Create an instance of the TestODBC_Class class
   TestODBC_Class  Example;
   ptr = strcat(ConnectStr,
         "CommitMode=2; ExtendedDynamic=1;DefaultPkgLibrary=DB2USER;");
   ptr = strcat(ConnectStr,
         "DefaultPackage=A/DEFAULT(IBM),2,0,1,0,512;");
   ptr = strcat(ConnectStr,
         "AllowUnsupportedChar = 1;");

   // Connect to DB2 for i database
   if (Example.ConHandle != NULL)
   {
```

```
   rc = SQLDriverConnect(Example.ConHandle, NULL,
   (SQLCHAR *)ConnectStr, SQL_NTS, NULL, 0, NULL, SQL_DRIVER_NOPROMPT);
```

Example 3: Specify the connection attribute keywords on the connection string

Here is an explanation of some of the keywords:

- **CommitMode:** Specifies the default transaction-isolation level. In this case, it is set to Read Uncommitted (*CHG). This would require journaling if it is used.
- **ExtendedDynamic:** Specifies whether to use an extended dynamic (package) support. A value of **1** enables the packages. This is the default for the IBM i ODBC driver.
- **DefaultPkgLibrary:** Specifies the library for the SQL package.
- **DefaultPackage:** Specifies how the extended dynamic (package) support behaves.

The SQL packages are server-side repositories for SQL statements. Packages contain the internal structures (such as parse trees and access plans) necessary to run SQL statements. Because SQL packages are a shared resource, the information built when a statement is prepared is available to all the users of the package. This saves processing time, especially in an environment where many users invoke the same or similar statements. Because SQL packages are permanent, this information is also saved across job termination and across system IPLs. In fact, SQL packages can be saved and restored on other systems. Usually, there is no need to delete SQL packages, but sometimes it is still reasonable to delete them, especially when you make significant changes to the database design. You can use the IBM i `WRKOBJ` command to locate and delete an SQL package. The following command is used to find the SQL package for the testodbc application:

```
   WRKOBJ OBJ(JAREK/*ALL) OBJTYPE(*SQLPKG)
```

The actual package name is TESTODBCVBA and is generated by the system. The SQL packages are named by taking the client application name and appending three letters that are an encoded set of the package configuration attributes.

Refer to the FAQ section of the *DB2 for i: Improving Performance with SQL Packages* website (at: **ibm.com**/systems/power/software/i/db2/support/tips/sqlperffaq.html) for details about the IBM i SQL package support.

Techniques for improving ODBC performance

The Extended Dynamic support was one of techniques used by the sample application to try and improve ODBC performance. Depending on your application architecture, this section covers other methods of improving ODBC performance. It is highly recommended that you attend the DB2 for i SQL performance workshop (at **ibm.com**/systems/power/software/i/db2/education/performance.html) to ensure that you have the required skills for tuning the performance of SQL-based applications such as Linux ODBC applications.

Reusable open data paths

An open data path (ODP) definition is an internal IBM i object that is created when certain SQL statements (such as `OPEN`, `INSERT`, `UPDATE`, and `DELETE`) are run for the first time in a given job (or connection). An ODP provides a direct link to the data so that I/O operations can occur. The process of creating a new ODP object is expensive from a performance perspective. Therefore, when possible, the DB2 for i run time tries to reuse the existing ODPs. Therefore, it is important that applications employ programming techniques that allow the DB2 run time to reuse ODPs.

With dynamic interfaces such as ODBC or JDBC, full opens are avoided by using a *prepare once, run many* programming paradigm. It means that when an SQL statement is going to run more than one time, you should prepare the statement just one time and then repeatedly run the prepared statement on later runs of the SQL statement. The code snippet in Example 4 illustrates how to implement the *prepare once, run many* programming technique.

```
strcpy((char *) SQLStmt,
       "select lstnam, city from qiws.qcustcdt where state = ?"); [1]
// Prepare the SQL statement
rc = SQLPrepare(Example.StmtHandle, SQLStmt, SQL_NTS); [2]
// Bind the columns in the result data set returned to
// application variables
rc = SQLBindCol(Example.StmtHandle, 1, SQL_C_CHAR, (SQLPOINTER)
                Example.LastName, sizeof(Example.LastName), NULL);
rc = SQLBindCol(Example.StmtHandle, 2, SQL_C_CHAR, (SQLPOINTER)
                Example.City, sizeof(Example.City), NULL);
for (int i = 0; i < LOOPS ; i++)
  {
    //Set the current value for the parameter marker
    rc=SQLBindParameter(Example.StmtHandle,
                        1,SQL_PARAM_INPUT,SQL_C_CHAR,SQL_CHAR,
                        3,0,(SQLPOINTER) currentState[i%3],0,NULL); [3]
    //Execute the statament with the current parameter
    rc = SQLExecute(Example.StmtHandle); [4]
    cout << endl << "Current State : " << currentState[i%3] << endl;
    // Display The Results Of The SQL Query
    Example.ShowResults();
    if(rc != 0)
        break;
  }
```

Example 4: Use the "prepare once, run many" programming technique

The error-handling code has been removed for clarity. At **[1]**, the statement text is assigned to a variable. Note that a parameter marker is used in the WHERE clause. The statement is then prepared just one time at **[2]**. The current value of the parameter is bound to the prepared statement at **[3]**. The prepared statement is run at **[4]**. Steps **[3]** and **[4]** are repeatedly run in a *for* loop.

Connection pooling

The unixODBC manager supports connection pooling, which can potentially limit the number of connections required by your application that accesses the IBM i database. This depends on whether your application processes that are running in a Linux partition can be shared or reused by multiple clients. The unixODBC connection pooling implementation is scoped to a process. In other words, the connection is reused if the calling process is the same as the process that initiated it. If your application creates and drops connections from a limited number of persistent processes, the connection pooling certainly improves performance. Setting up the connection pooling is easy — edit the odbcinst.ini file so that it looks similar to Figure 7.

```
[ODBC]
Trace        = No
Trace File   = /tmp/sql.log
Pooling      = Yes

[IBM i Access ODBC Driver]
Description   = IBM i Access for Linux ODBC Driver
Driver = /opt/ibm/iSeriesODBC/lib/libcwbodbc.so
Setup  = /opt/ibm/iSeriesODBC/lib/libcwbodbcs.so
Threading     = 2
FileUsage     = 1
DontDLClose = 1
CPTimeout    = 600
```

Example 5: Set up connection pooling by editing the odbcinst.ini file

The critical values are `Pooling = Yes` and `CPTimout`. You might want to experiment with the latter value. In the example, it was set to 10 minutes. The settings shown in Example 5 work fine with a test program that opens and closes a connection several times. When the program is run, there is one QZDASOINIT job on the IBM i because the SQLDisconnect request is intercepted by the driver manager, and the connection is actually not closed.

Notice that the connection pooling implemented by unixODBC is functionally similar to the connection pooling on Microsoft Windows® provided by the Microsoft ODBC manager.

Blocking, stored procedures, and result sets

You can reduce the number of trips to the server with blocking of inserts and fetches, stored procedures, and result sets. For specific DB2 for i implementation details, refer to the IBM i Information Center.

Troubleshooting

Troubleshooting a multitier application might be a bit tricky because you need to deal with software components that reside in separate partitions. The experience at IBM shows that two tools are particularly useful for pinning down the potential problem areas:

- Job log messages for an IBM i server job usually provide sufficient details to isolate issues.
- The ODBC trace utility provides granular information about the data flow between your Linux application and the DB2 for i server.

DB2 engine feedback

As mentioned, an ODBC client communicates with a corresponding IBM i job. This server job runs the SQL requests on behalf of the client. More precisely, when an ODBC client submits an SQL statement, the statement is passed to a server job that, in turn, calls DB2 for i to run the statement. The results are then reformatted and marshaled to the client. The IBM i database server jobs are called QZDASOINIT, and they run in the QUSRWRK subsystem. At any given time, there might be a large number of database server jobs active on the system. The first step is to identify the job that serves your particular ODBC connection. The easiest method to accomplish this task is to run the following IBM i CL command:

```
WRKOBJLCK OBJ(DB2USER) OBJTYPE(*USRPRF)
```

Here, DB2USER is the user profile you use to connect to the IBM i platform. Notice that a QZDASOINIT job is assigned to an ODBC connection *after* the connection has been established. So, you need to set a breakpoint in the client application after the SQLDriverConnect API call.

When the Work with Object Locks screen appears, type **5** (Work with Job) next to the only QZDASOINIT job listed. This shows the Work with Job dialog box. Select option **10** to display the job log for the database server job. Now, you can search the job log for error messages generated by DB2 for i or the operating system.

Sometimes, it is also useful to activate the Database Monitor to collect a detailed DB2 for i SQL trace. Database monitor collections can be helpful to diagnose both functional and performance issues.The easiest way to collect monitor data for a Linux ODBC connection is to specify the **TRACE=2** keyword on the connection string. Here is an example:

```
Char ConnStr[512]="DSN=db2iserver1;UID=db2user;PWD=userpwd;TRACE=2;";
```

You can find more details on the Database Monitor in the IBM Redbooks® titled, *SQL Performance Diagnosis on IBM DB2 UDB for iSeries* (SG24-6654-00).

ODBC trace

The IBM i ODBC driver provides the `cwbtrc` utility, which can be used to collect detailed client-side traces and logs. Rather than writing all the data to a daemon and letting it format and store the data (which is how the Windows tracing operates), the Linux tracing writes everything to ASCII text files.

The tracing is switched off by default. The following parameters are recommended when invoking the cwbtrc utility: `/opt/ibm/iSeriesAccess/bin/cwbtrc /dt:1 /hl:1`

The /dt:1 parameter turns the detail trace on and /hl:1 turns the history log on. At this point, tracing is activated and no further configuration is required. The trace files are placed in the .iSeriesODBC directory that is located in your home directory. The output files have the following naming convention:

- cwbdetail-<process name>-pid.csv
- cwbhistory-<process name>-pid.csv

For specific syntax and functionality of other `cwbtrc` parameters, refer to the IBM i Information Center documentation

The detail trace and the history log store their data in semi-colon delimited text files. Example 6 shows a short excerpt from a sample detail trace that shows that the SQLDisconnect call failed because the connection had a pending transaction.

```
04/07/2013;00:57:58.134;ODBC;21748;21748;10008;0x82000001: odbcconn.SQLDisconnect
Entry
04/07/2013;00:57:58.134;ODBC;21748;21748;10008;odbcerr.storeError Entry
04/07/2013;00:57:58.135;ODBC;21748;21748;10008;odbcerr.finishAndInsertErr Entry
04/07/2013;00:57:58.135;ODBC;21748;21748;10009;err: [IBM][iSeries Access ODBC
          Driver]Invalid transaction state. dsn: db2iserver1 sys: os400
04/07/2013;00:57:58.135;ODBC;21748;21748;10008;odbcerr.finishAndInsertErr Exit
04/07/2013;00:57:58.135;ODBC;21748;21748;10008;odbcerr.storeError Exit
04/07/2013;00:57:58.136;ODBC;21748;21748;10008;0x82000001: odbcconn.SQLDisconnect Exit
rc=-1
```

Example 6: Sample Detail Trace showing that the SQLDisconnect call has failed

Using the IBM i JDBC driver

The IBM i operating system also includes a JDBC driver to enable DB2 for i data access from Java™ applications running on Linux. The IBM Toolbox for Java library provides a JDBC driver which is a *Type 4* driver optimized for accessing DB2 for i databases. The Toolbox does not require additional client support over and above what is provided by the Java virtual machine (JVM) and TCP/IP. The JDBC driver itself uses a native IBM i protocol to communicate with the back-end database server job.

JTOpen is the open-source version of the IBM Toolbox for Java offering that serves as a fast-path mechanism to make fixes and enhancements available to developers.

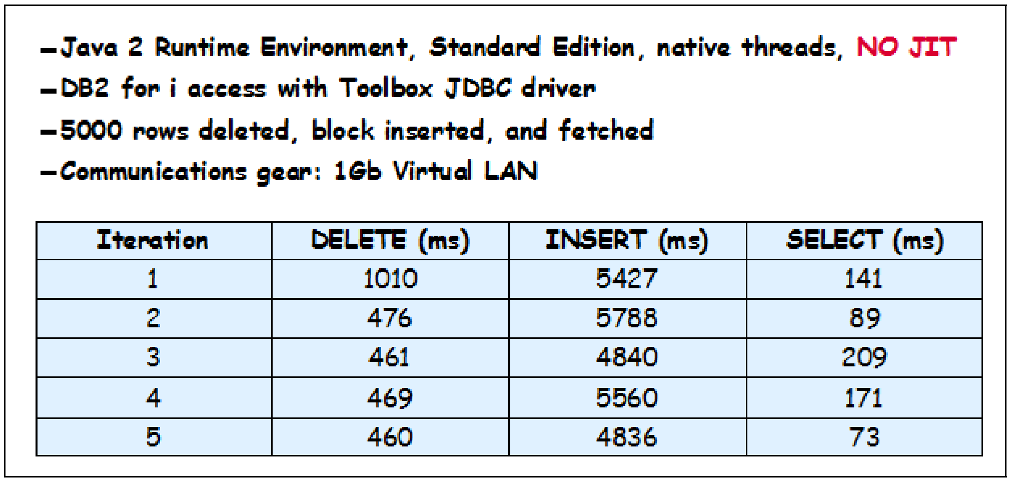
The Toolbox jt400.jar file contains the classes for the JDBC driver. You can find client installation instructions for the JDBC driver at the IBM i Information Center.

Initial configuration considerations

Practical experience shows that the JDBC application performance greatly depends on the programming techniques and the type of SQL being run against the server. However, significant performance gains can be achieved by tweaking the Linux partition environment. This section covers the two configuration options: Java runtime environment (JRE) and virtual or physical LAN.

Java runtime environment

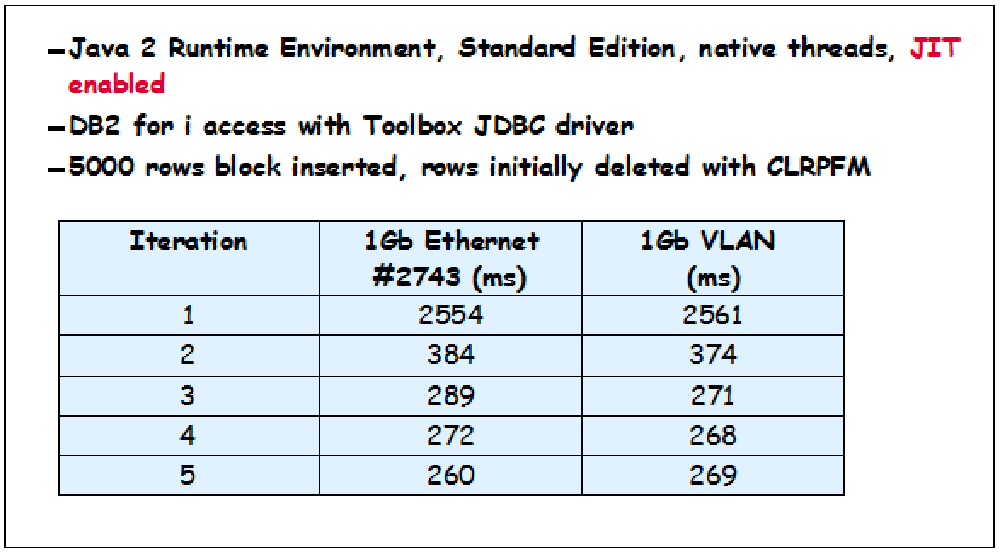
Although JRE is usually preinstalled in a Linux partition, it is strongly recommended that the latest IBM SDK for Java (64-bit) is downloaded from the IBM Java support website. The JRE implements the state-of-the-art, just-in-time (JIT) compiler that can dramatically improve performance of a Java application, as illustrated in Figure 3 and Figure 4. In both cases, a simple Java program runs in a loop.



- Java 2 Runtime Environment, Standard Edition, native threads, NO JIT
- DB2 for i access with Toolbox JDBC driver
- 5000 rows deleted, block inserted, and fetched
- Communications gear: 1Gb Virtual LAN

Iteration	DELETE (ms)	INSERT (ms)	SELECT (ms)
1	1010	5427	141
2	476	5788	89
3	461	4840	209
4	469	5560	171
5	460	4836	73

Figure 3: Java application performance, no JIT



–Java 2 Runtime Environment, Standard Edition, native threads, JIT enabled

–DB2 for i access with Toolbox JDBC driver

–5000 rows block inserted, rows initially deleted with CLRPFM

Iteration	1Gb Ethernet #2743 (ms)	1Gb VLAN (ms)
1	2554	2561
2	384	374
3	289	271
4	272	268
5	260	269

Figure 4: Java application performance with JIT-enabled

Virtual or physical LAN

One of the most significant benefits of installing Linux in an IBM i partition is the ability to use virtual LAN and virtual disk storage. This usually results in reduced hardware costs through improvements in data-transfer performance and enhanced disk protection. The performance tests conducted by IBM show that the virtual 1Gb LAN delivers data throughput that is similar to the dedicated 1 Gb Ethernet adapters (model 2743) that are connected with a point-to-point multimode fiber. The test configuration is illustrated in Figure 5.

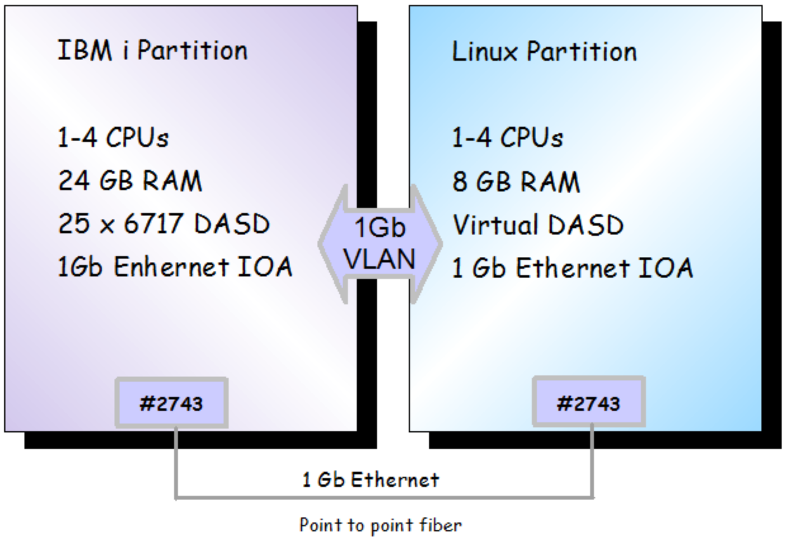
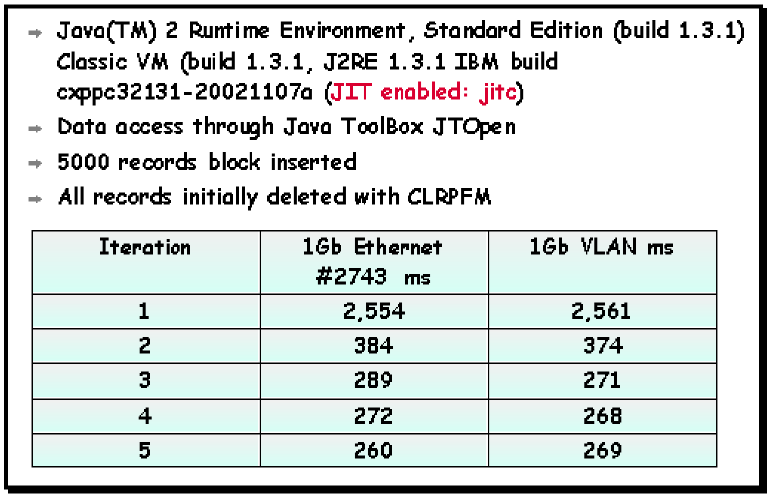


Figure 5: VLAN compared with a 1 Gb Ethernet input/output adapter (IOA) configuration

Figure 6 shows the test results measured for a sample Java application.



- Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1) Classic VM (build 1.3.1, J2RE 1.3.1 IBM build cxppc32131-20021107a (JIT enabled: jitc)
- Data access through Java ToolBox JTOpen
- 5000 records block inserted
- All records initially deleted with CLRPFM

Iteration	1Gb Ethernet #2743 ms	1Gb VLAN ms
1	2,554	2,561
2	384	374
3	289	271
4	272	268
5	260	269

Figure 6: VLAN compared with a 1 Gb Ethernet IOA throughput

Other real-life database-access scenarios tested for solution providers' solutions show that similar performance can be expected for VLAN and 1Gb Ethernet (model 2743). However, the VLAN configuration has the following two advantages over a physical LAN connection.

- No additional communications gear is required.
- It is a dedicated point-to-point connection, so there is no interference from other devices on the same network segment.

Alternatively, certain application scenarios might benefit from a physical LAN adapter that is dedicated to a Linux partition. For instance, in a three-tier application architecture, clients connect to an application server that is running in a Linux partition. The application server, in turn, connects to DB2 for i where data is stored and retrieved. In this case, configuring a separate physical LAN connection in a Linux partition has the following advantages:

- An unnecessary additional hop is eliminated from the network route.
- Client-application server traffic does not unnecessarily saturate IBM i communications.

Techniques for improving JDBC performance

This section covers JDBC implementation details for a sample Java program called TestJDBC (the source is provided at the end of this paper in Appendix A). You can find many similarities with the performance-tuning methods discussed previously with the ODBC driver.

The application uses the Java Naming and Directory Interface (JNDI) to get a database connection, delete 5000 rows, generate and insert 5000 rows and, finally, retrieve the newly-inserted rows from an SQL table called COFFEES. The application runs in a Linux partition and accesses DB2 for i through the JTOpen driver. Figure 7 shows the software components involved in running this simple application.

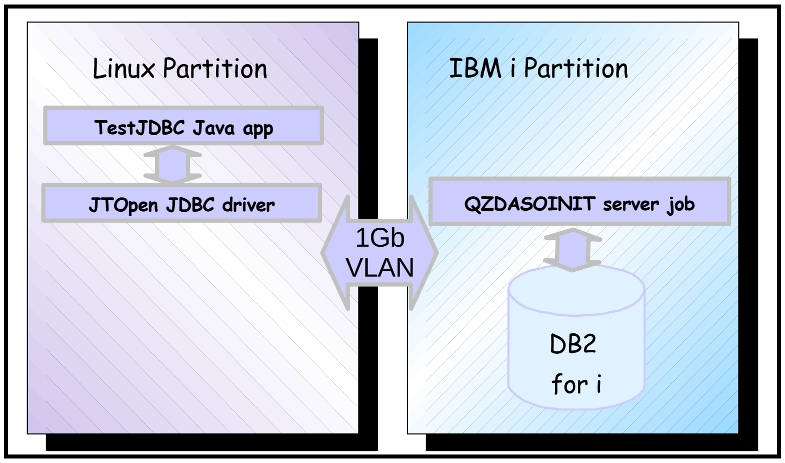


Figure 7: JDBC client accessing DB2 for i

Apart from the JTOpen driver, you might need additional JAR files depending on your version of Java Development Kit (JDK) to successfully compile and run the JDBCTest application in a Linux partition. These JAR files are providerutil.jar and fscontext.jar and they can be accessed by downloading a simple file system JNDI provider.

The location of the JAR files needs to be added to the `CLASSPATH` environment variable. For instance, the following entry can be added to the Linux .profile script:

```
CLASSPATH=.:/home/code/JTOpen/jt400.jar
CLASSPATH=$CLASSPATH:/home/code/fscontext/providerutil.jar
CLASSPATH=$CLASSPATH:/home/code/fscontext/fscontext.jar
export CLASSPATH
```

To compile the source, use the following command in the Linux partition:

```
javac TestJDBC.java
```

Although the TestJDBC application is fairly simple, it illustrates several important aspects of efficient JDBC programming for DB2 for i.

Connection pooling

A JDBC connection can be obtained either through the `DataSource` object or the `DriverManager` object. Both methods consume a significant amount of system resources. A fairly large amount of system resources is required to create and maintain the connection and then release it when it is no longer needed. JTOpen allows you to establish a pool of database connections that can be shared by multiple client applications. Connection pooling can improve response time because the connection-pool manager can locate and use an existing connection. When the database request is complete, the connection returns to the connection pool for reuse.

In the JDBCTest sample, JNDI is first used to register a connection pool-capable data source. Example 7 is an excerpt from the `deployDataSource` method of the sample application.

```
Hashtable<String, String> env = new Hashtable<String, String>();
env.put(Context.INITIAL_CONTEXT_FACTORY,
       "com.sun.jndi.fscontext.RefFSContextFactory");
Context ctx = new InitialContext(env); [1]

// Register an AS400JDBCConnectionPoolDataSource object.
AS400JDBCConnectionPoolDataSource tkcpds = new
AS400JDBCConnectionPoolDataSource(); [2]
tkcpds.setServerName(serverName) ;
tkcpds.setDatabaseName(databaseName) ;
tkcpds.setUser(userName) ;
tkcpds.setPassword(password) ;
tkcpds.setDescription("Toolkit Connection Pooling DataSource object");
…
ctx.rebind("jdbc/ToolkitDataSource", tkcpds); [3]
```

Example 7: Excerpt from the deployDataSource method that registers a connection pool-capable data source

At **[1]**, the initial JNDI context is created. Then at **[2]**, an *AS400JDBCConnectionPoolDataSource* object is instantiated. This is the JTOpen implementation of the *javax.sql.ConnectionPoolDataSource* interface. An *AS400JDBCConnectionPoolDataSource* object supports a number of methods that can be used to set specific System i properties of the underlying *DataSource* object. Some of these properties are discussed in greater detail in the following sections of this paper. At **[3]**, the data source object is bound to an arbitrary name that is used in the main application. Note that the *deployDataSource* method needs to be run just one time, unless some of the data source properties need to be changed. In that case, you need to rebind. The JNDI-registered data source is then used in the main application to initialize the connection pool (refer to Example 8).

```
AS400JDBCConnectionPoolDataSource tkcpds = [1]
(AS400JDBCConnectionPoolDataSource)ctx.lookup("jdbc/ToolkitDataSource")
;
// Create an AS400JDBCConnectionPool object.
AS400JDBCConnectionPool pool = new AS400JDBCConnectionPool(tkcpds);[2]
// Adds 1 connection to the pool that can be used by the application
//(creates the physical database connections based on the data source).
pool.fill(1);  [3]
…
con = pool.getConnection(); [4]
```

Example 8: Initializing the connection pool from the main application

At **[1]**, the JNDI service is used to instantiate *AS400JDBCConnectionPoolDataSource*. Then at **[2]**, an *AS400JDBCConnectionPool* object is created. This object is responsible for managing the connection pool. At **[3]**, the connection pool is initialized with one connection. The pool object is used at **[4]** to obtain a database connection. There are several additional things to understand about this code snippet.

- The advantage of using JNDI to register the data source object (as opposed to merely instantiating it in the main method) is that, in a more realistic scenario, the data source is registered just one time (at the application deployment time). Additionally, the data source properties do not have to be hard coded in the main method and can be changed at any time and then rebound without affecting the application.
- *AS400JDBCConnectionPool* is a proprietary JTOpen implementation of a connection pool. Specifically, it does not implement the referenceable interface and, thus, cannot be bound through JNDI services.
- The JTOpen *AS400JDBCConnectionPoolDataSource* does not support statement caching as described in the JDBC 3.0 specification. Statement caching is implemented through SQL packages. This feature is covered in the following section.

Note: The native JDBC driver for the IBM i operating system supports both connection pooling and statement caching in a JDBC-compliant manner.

Extended dynamic SQL and statement caching

As mentioned, the JTOpen data source object supports a number of properties that can be used to fine tune the database performance. In this section, the properties used to enable extended-dynamic SQL support and local-statement caching is discussed. Example 9 is a relevant code excerpt from the *deployDataSource* method.

```
AS400JDBCConnectionPoolDataSource tkcpds = new
AS400JDBCConnectionPoolDataSource();
…
tkcpds.setExtendedDynamic(true);[1]
tkcpds.setPackage("JDBCDS"); [2]
tkcpds.setPackageCache(true); [3]
tkcpds.setPackageCriteria("select"); [4]
```

Example 9: Code excerpt from the deployDataSource method

At **[1]**, the extended dynamic SQL support is enabled. Note that the default is disabled for JTOpen. The extended dynamic SQL function is often referred to as *SQL package support*. The SQL packages are server-side repositories for SQL statements. Packages contain the internal structures, such as parse trees and access plans that are needed to run the SQL statements. Because SQL packages are a shared resource, the information built when preparing a statement is available to all users of the package. This saves processing time, especially in an environment where many users use the same or similar statements. Also, because SQL packages are permanent, this information is saved across job initiation and termination and during IPLs. In fact, SQL packages can be saved and restored on other servers.

At **[2]**, the SQL package is named. Note that the actual package name on the IBM i model is JDBCDSBAA. The package name is generated by taking the name specified on the client and appending three characters that are an encoded set of the package configuration attributes. By default, an SQL package does not contain unparameterized Select statements. This behavior is overridden at **[4]** by setting the *PackageCriteria* property to ***select***.

At **[3]**, the local statement cache is enabled. A copy of the SQL package is cached on the Linux client. This might improve application performance in a way that is similar to JDBC 3.0 statement caching. However, notice that local-statement caching is not recommended for large SQL packages. The copy of an SQL package is cached when the database connection is obtained. This might cause increased network traffic and also slow down the connection setup time.

Reusable ODPs

As described in the ODBC section of this paper, it is very important from a performance perspective that the applications employ programming techniques that allow the DB2 run time to reuse ODPs.

With the JDBC interface, the most efficient way to avoid full opens is to employ the *prepare once, run many* programming paradigm. Ideally, an SQL statement that is run more than one time must be prepared just one time — for example, in the class constructor — and then reused for the consecutive executions. On the first run, the ODP that is created for a given statement is associated with the appropriate entry in the statement cache. On consecutive executions, the prepared statement and the corresponding ODP can be quickly located in the statement cache and reused. The code snippet in Example 10 illustrates the *prepare once, run many* programming technique.

```
PreparedStatement pstmt = con.prepareStatement("INSERT INTO COFFEES
VALUES( ?, ?, ?, ?, ?)");[1]
…
for (int i = 0; i < outerNumOfLoops; i++) {
    // insert
    for (int j = 0; j < numOfLoops; j++) {
        pstmt.setString(1, "Lavaza_" + Integer.toString(j));
        …
        pstmt.addBatch();
        }

    int [] updateCounts = pstmt.executeBatch();[2]
    con.commit();
}
```

Example 10: Code snippet that illustrates the "prepare once, run many" programming technique

At **[1],** a *PreparedStatement* `pstmt` object is instantiated for a parameterized INSERT statement. The statement is prepared just one time. The *PreparedStatement* object is then repeatedly used at **[2]** to perform blocked inserts.

Block inserts

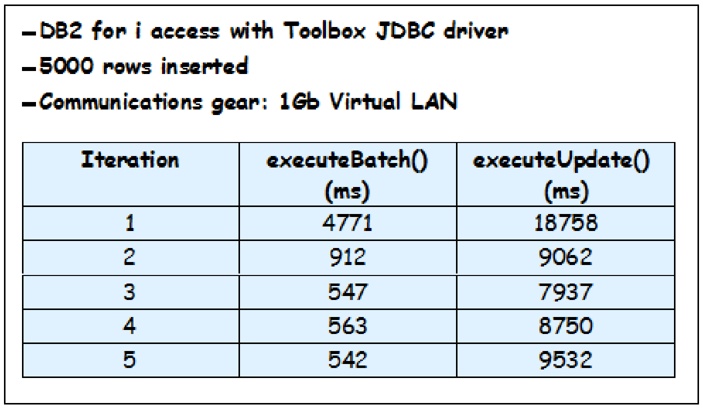
Occasionally, you might need to populate a table initially or insert a modified result set into a temporary table. The efficiency of the insert operation for a large set of records can be improved dramatically by using the *executeBatch* method rather than the *executeUpdate* method. The JTOpen driver converts parameterized INSERT statements in an *executeBatch* method to a block-insert statement. This significantly reduces the data flow and system resources required to perform the insert. The following code in Example 11 shows how to take advantage of the *executeBatch* method.

```
PreparedStatement pstmt =
con.prepareStatement("INSERT INTO COFFEES VALUES( ?, ?, ?, ?, ?)"); [1]
for (int j = 0; j < numOfLoops; j++) {
  pstmt.setString(1, "Lavaza_" + Integer.toString(j));
  pstmt.setInt(2, i);
  pstmt.setFloat(3, 4.99f);
  pstmt.setInt(4, 0);
  pstmt.setInt(5, 0);
  pstmt.addBatch(); [2]

 }
int [] updateCounts = pstmt.executeBatch(); [3]
```

Example 11: Excerpt from the deployDataSource method

At **[1]**, a parameterized INSERT statement is prepared. Note that all the columns must be parameterized. Otherwise, the *executeBatch* method might throw an SQL error. In other words, you must not mix parameter markers with literals or special registers in the *VALUE* clause. At **[2]**, the *addBatch* method is used to add a new record to the client-record buffer. At **[3]**, all locally buffered data is sent to the server that, in turn, performs a block insert.

Figure 20 shows the performance data for the *executeBatch* and *executeUpdate* methods.



–DB2 for i access with Toolbox JDBC driver
–5000 rows inserted
–Communications gear: 1Gb Virtual LAN

Iteration	executeBatch() (ms)	executeUpdate() (ms)
1	4771	18758
2	912	9062
3	547	7937
4	563	8750
5	542	9532

Figure 8: Comparing the performance of executeBatch() and executeUpdate()

Troubleshooting

Troubleshooting a multitier application might be a bit tricky because you need to deal with software components that reside in separate partitions. Some tools are particularly useful for pinning down the potential problem areas:

- Joblog messages for a database server job provide sufficient details to isolate DB2 for i runtime issues.
- Database monitor provides a detailed trace of SQL requests processed by DB2.
- JDBC trace utility provides granular information about the data flow between the Linux application and the IBM i job.

DB2 engine feedback

The techniques documented in the DB2 engine feedback of the ODBC driver can also be applied with the IBM i JDBC driver to collect feedback from the DB2 for i engine.

JDBC Trace utility

The JTOpen driver provides a tracing utility that can be used to collect detailed client-side traces. The JDBC tracing is turned off by default. It can be enabled by setting the *trace* property to **`true`**. Here is an example of how to switch on tracing using the *setTrace* method on the data source object:

```
tkcpds.setTrace(true);
```

The trace can also be enabled when using the *DriverManager.getConnection()* method for obtaining a connection to DB2 for i. In this case, you can add the *trace* property to the connection string. This approach is illustrated in the following code snippet.

```
Class.forName("com.ibm.as400.access.AS400JDBCDriver"); [1]
…
con = DriverManager.getConnection("jdbc:as400://teraplex;trace=true;",
                                  "db2user","db2pwd"); [2]
```

Example 12: Enabling a JDBC trace

At **[1]**, an instance of the JTOpen JDBC driver is initiated. Then at **[2]**, the driver is used to obtain a connection to the server. Note how the trace property is added to the database URL. Refer to the IBM Toolbox for Java JDBC properties documentation on the IBM i Information Center website for a complete list of properties supported by the JTOpen driver.

Using the DB2 Connect middleware

DB2 Connect provides the capability of accessing data stored in DB2 for i and DB2 for z/OS® databases. DB2 Connect offers capabilities to access DB2 family members through several SQL interfaces (for example, JDBC and ODBC), gateway functions (such as connection pooling) and federated database functionality. Many applications that support the DB2 family of products use DB2 Connect as a key middleware component. DB2 Connect is offered as a separate license product.

- IBM DB2 Connect Personal Edition
- IBM DB2 Connect Enterprise Edition
- IBM DB2 Connect Application Server Edition
- IBM DB2 Connect Unlimited Edition for System i

In reality, the full DB2 Connect product does not need to be installed in order for a Linux application to interoperate with DB2 for i databases. The actual DB2 middleware drivers can be downloaded and installed separately using one of the IBM drivers and clients. This white paper uses the IBM Data Server Driver Package for its examples. The IBM Data Server Driver Package is a lightweight solution and the best package for end user code deployment. It provides robust runtime support for applications using ODBC, CLI, .NET, OLE DB, PHP, Ruby, JDBC, or SQLJ without the need of installing IBM Data Server Runtime Client or IBM Data Server Client.

The only reason that it is necessary to purchase a DB2 Connect product license is for the license file that enables the DB2 Connect drivers to access DB2 for i databases. For production usage, the DB2 Connect Unlimited Edition for System i packaging typically offers the best terms for IBM i customers. Contact your local IBM representative or business partner for pricing information. For more information on this product, refer to: **ibm.co**m/software/data/db2/db2connect/edition-uei.html. A trial DB2 Connect license file for evaluation purposes can be obtained by sending an email to: rmahendr@us.ibm.com.

Given that the IBM i ODBC and JDBC drivers are provided at no additional charges, you might be wondering why would an IBM i client ever purchase a DB2 Connect product license. Here are several possible scenarios that require usage of the DB2 Connect in a Linux partition.

- Linux application uses embedded SQL interface to access the DB2 for i server.
- A single SQL interface is needed to access different back-end DB2 or IBM Informix® servers.
- Specific DB2 Connect functions such as a distributed join (for example, a federated database), is needed between DB2 and Informix servers.

DB2 Connect environment overview

DB2 Connect uses the Distributed Relational Database Architecture (DRDA) standard from The Open Group to reduce the cost and complexity of accessing data that is stored in DB2 for i and other DRDA-compliant database servers. DRDA defines the rules, the protocols, and the semantics for accessing distributed data. Applications can access data in a distributed relational environment by using SQL statements. In a distributed environment, the system that runs the application and sends the SQL requests across the network is called an application requester (AR). A remote database server that runs SQL requests that are submitted by an application requester is an application server (AS). In the example shown in Figure 9, DB2 Connect plays the role of an application requester; and DB2 for i is an application server.

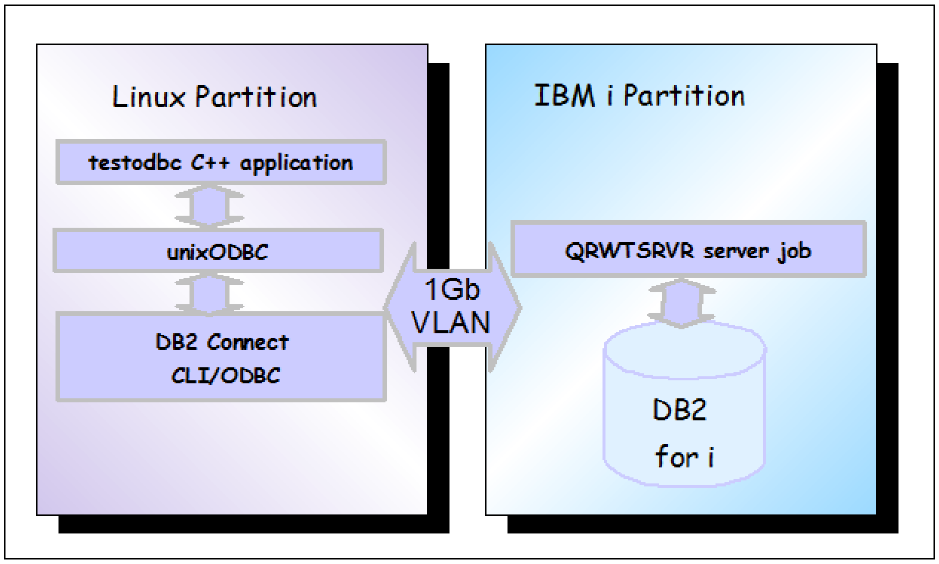


Figure 9: Using DB2 Connect to access IBM i data

In this case, DB2 Connect uses a TCP/IP connection with the IBM i system. The IBM i DRDA application server implementation is based on multiple connection-oriented server jobs that run in the QUSRWRK subsystem. A DRDA listener job (QRWTLSTN) listens for TCP connect requests on port 446. After an application requester connects to the listener job, the listener wakes up one of the QRWTSRVR prestart jobs and assigns it to a given client connection. Any further communication occurs directly between the client application and the assigned QRWTSRVR job.

Configuring the IBM i

The following configuration steps are needed to ensure that your IBM i system can be accessed with a DB2 Connect driver.

1. Verify that the TCP/IP stack works correctly. It is assumed that the virtual LAN is used to communicate with the IBM i partition. First, obtain the IBM i system's IP address (or host name) and ping the IBM i system from the Linux partition.
2. Determine the name of the local database entry in the IBM relational database directory. This name can be displayed by using the `DSPRDBDIRE` (Display Relational Database Entries) command. Record the entry name with a location value of `*LOCAL`. In this example, the name is `mysystem` and the TCP/IP host name is also `mysystem`. For simplicity, it is recommended that the TCP/IP host name and local entry be the same value. In the rare case where a `*LOCAL` entry does not exist, use the `ADDRDBDIRE` (Add Relational Database Directory Entry) command. For example, the following command adds an entry named `mysystem`.
   ```
   ADDRDBDIRE RDB(MYSYSTEM) RMTLOCNAME(*LOCAL)
   ```
3. Create a schema called NULLID. This is necessary because the utilities included with DB2 Connect and DB2 for Linux store their packages in the NULLID schema. Because it does not exist by default on the IBM i system, you must create it using the following command:

```
CRTLIB LIB(NULLID)
```

4. Products that support DRDA automatically perform any necessary code-page conversions at the receiving system. For this to happen, both systems need a translation table from their code page to the partner code page. The default coded character set identifier (CCSID) on the IBM i platform is 65535, so a good number of DB2 for i tables have character columns defined with a CCSID of 655535. Because DB2 Connect does not have a translation table for this code page, the drivers will be unable to translate the data to an encoding that can be consumed by the Linux application. One way to get the data properly translated is to change the user profile that will be used on a DB2 Connect connection to contain a CCSID that can be properly converted. If US. English is stored in the character columns, the CCSID value of 037 can be used. The following command changes the CCSID for an individual user profile to 037:

   ```
   CHGUSRPRF db2user CCSID(37)
   ```

5. Verify that the default port 446 for DRDA service is used. To do this, go to the **Configure TCP/IP** menu (CFGTCP), select **Configure Related Tables**, and then select **Work with service table entries**. Verify that the DRDA service is set for port 446.

6. The Distributed Data Management (DDM) job must be started for DRDA to work. If you want the DDM job to be started automatically whenever TCP/IP is started, change the attributes of the DDM job by using the `CHGDDMTCPA` command and set the Autostart server parameter to *YES. If the system administrator chooses not to automatically start the server, issue the following command to start the DDM server job:

   ```
   STRTCPSVR SERVER(*DDM)
   ```

Installing the Data Server Driver Package

You can download the IBM Data Server Driver Package from the following link:
ibm.com/support/docview.wss?rs=4020&uid=swg27016878

Detailed installation instructions are document in the IBM DB2 Information Center at the following link:
https://ibm.biz/BdRzv6

Perform the following steps to install this package in the Red Hat 64-bit environment used in this paper:

1. Run the `installDSDriver` command to the extract the ODBC and JDBC drivers. This command also creates the db2profile script file that is needed for the Red Hat Bash shell environment.

2. Set the IBM Data Server Driver environment by running the db2profile script with the following command: `. db2profile`

Using the Data Server Driver ODBC support

The IBM Data Server Driver Package enables ODBC-based access of DB2 for i with its ODBC driver and DB2 call level interface (CLI) support. DB2 CLI is based on the Microsoft ODBC specification and the International Standard for SQL/CLI. These specifications were chosen as the basis for the DB2 CLI in an effort to follow industry standards and to provide a shorter learning curve for those application programmers already familiar with either of these database interfaces. In addition, some DB2-specific extensions have been added to help the application programmer to specifically use the DB2 features.

The DB2 CLI driver also acts as an ODBC driver when loaded by an ODBC driver manager. In contrast, the IBM i Access ODBC drivers can only be used when loaded by an ODBC driver manager. Usage of an ODBC driver manager enables an application to interact with multiple types of database servers.

Configuring a DB2 for i connection

The next step is updating the IBM Data Server Driver configuration file (db2dsdriver.cfg in the …/dsdriver/cfg subdirectory) with the attributes of the DB2 for i database that you want to access. This file contains the database directory information and configuration parameters for the database servers that need to be accessed by your applications.

You can use the db2dsdriver configuration file with ODBC, CLI, embedded SQL applications, .NET, OLE DB, PHP, or Ruby drivers. It is recommended that you use the db2dsdriver.cfg file instead of the db2cli.ini file because it supports a wide variety of drivers (the db2cli.ini file only supports ODBC/CLI access). Creating and populating the db2dsdriver.cfg configuration file is not required for these drivers. The applications can function without this configuration file. However, instead of specifying detailed information about the database name, host, port, and configuration parameters in your applications, you can use the configuration file to define aliases.

As shown in the following configuration file in Example 13, the db2dsdriver configuration file is an XML file. The DB2 for i server that is being accessed needs both a data source name (DSN) and a database entry. The name, host, and port attributes are the keys that tie together a DSN and database entry. The alias name for the DSN entry in this example is `db2isrvr` (and you can enter any value for the DSN entry). The remaining DSN entry attributes come directly from your IBM i server. The name attribute, `mysystem`, is the name of the local database directory that was identified in the *Configuring the IBM i* section. The port value of 446 is required because that is the TCP/IP port used by the DRDA protocol to communicate with IBM i.

```
<configuration>
   <dsncollection>
      <dsn alias="db2isrvr" name="mysystem" host="mysystem.mydomain.com"
           port="446">
      </dsn>
   </dsncollection>
   <databases>
       <database name="mysystem" host="mysystem.mydomain.com" port="446">
            <parameter name="CurrentSchema" value="MYSCHEMA" />
       </database>
   </databases>
   <parameters>
   </parameters>
</configuration>
```

Example 13: A sample db2dsdriver configuration file

The attributes of database entry duplicate the name, host, and port attributes of the DSN entry. In this example, an optional current schema parameter is defined for the DB2 for i database.

Earlier versions of DB2 for Linux, UNIX and Windows and DB2 Connect included a graphical interface known as Client Configuration Assistant for building a database entry for DB2 for i servers. “Appendix C: Client Configuration Assistant steps” documents how to use this graphical wizard with DB2 for i databases.

The final configuration step for a DB2 for i database is applying the DB2 Connect license file. Multiple DB2 for i databases can be accessed as soon as the DB2 Connect license file has been applied to your Linux installation. For this installation, the DB2 Connect license was applied by placing the license file in the /root/dsdriver/license directory.

Testing the ODBC connectivity

Before the DB2 CLI driver can be used as a ODBC driver by the application running on a Linux server, a PowerPC version of the unixODBC driver manager must be installed.

The `isql` command provides an easy way to test out ODBC connectivity through the unixODBC driver manager. A couple of configuration steps are necessary before the `isql` utility can be used. First, the Data Server Driver needs to be registered with the unixODBC driver manager by adding the driver information to the odbcinst.ini configuration file. For this installation, this file was located in the /etc directory. Here is an example of the content within the odbcinst.ini file:

```
[DB2_DSD]
Description = DB2 Data Server driver
Driver      = /root/dsdriver/lib/libdb2o.so
fileusage   = 1
DontDLClose = 1
```

The next step in the configuration process is to add a DSN entry to the odbc.ini configuration file for the DB2 for i database that needs to be accessed. This can be accomplished by adding the following entry to the odbc.ini configuration file. The DSN entry name, db2isrvr, matches the alias value defined for the DSN defined in the db2dsdriver.cfg file.

```
[db2isrvr]
Description = DB2 DS driver
Driver = DB2_DSD
```

With this configuration in place, the ODBC connection can be tested with the following isql command:

```
isql db2isrvr db2user userpwd
```

This isql invocation includes the user ID and password because those values were not specified in any of the configuration files.

Testing CLI connectivity

The IBM Data Server Driver installation provides a db2cli command in the bin subdirectory which can be used to test CLI connectivity to a DB2 for i back-end server. Before using the `db2cli` command, you need to verify that the prerequisite IBM XL C/C++ for the Linux compiler is installed on your Linux system. If not, this compiler needs to be installed before using the `db2cli` utility.

The following example shows how the `db2cli` command invocation references the DSN entry, db2isrvr, defined in your db2dsdriver.cfg file. In addition, the IBM i user profile and password must be specified so that the utility can authenticate with the specified IBM i system.

```
db2cli validate -dsn db2isrvr –connect –user db2user –passwd userpwd
```

The `db2cli` command displays the specified DSN value along with the connection test results. Example 14 contains selected output from the `db2cli` command.

```
...
===========================================================================
db2dsdriver.cfg validation for data source name "db2isrvr":
===========================================================================
[ Parameters used for the connection ]

Keywords                  Valid For     Value
---------------------------------------------------------------------------
DATABASE                  CLI,.NET,ESQL mysystem
HOSTNAME                  CLI,.NET,ESQL mysystem.mydomain.com
PORT                      CLI,.NET,ESQL 446
CURRENTSCHEMA             CLI,.NET      MYSCHEMA

===========================================================================
Connection attempt for data source name "db2isrvr":
===========================================================================

[SUCCESS]

===========================================================================
The validation is completed.
===========================================================================
```

Example 14: db2cli command output

Using the Data Server Driver JDBC support

The IBM Data Server Driver Package actually includes two JDBC drivers. One of the drivers is known as the Universal JDBC driver. The Universal JDBC driver is a *Type 4* driver meaning that its implementation is pure Java. The other driver is known as the CLI legacy driver because it is a *Type 2* JDBC driver that requires the DB2 CLI support for accessing back-end DB2 databases. IBM recommends the usage of the Universal JDBC driver because it is more strategic and simpler to deploy.

The Universal JDBC driver uses the same DRDA protocol discussed in the “DB2 Connect environment overview” section. The deployment environment for a Java application running on Linux that accesses DB2 for i databases with the Universal JDBC driver is illustrated in Figure 10.

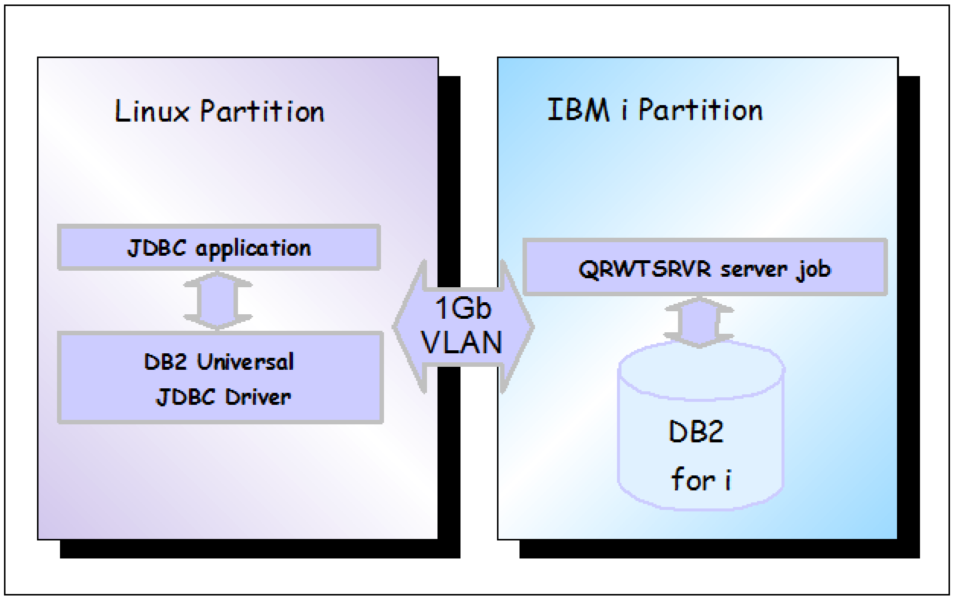


Figure 10: Accessing DB2 for i with DB2 Universal JDBC driver

Installation and configuration

The JDBC driver is installed as part of the IBM Data Server Driver Package installation process that was reviewed earlier. The IBM Data Server Driver installation does not address the prerequisite JDK. The Universal JDBC driver requires JDK 1.4.2 or later. Usage of JDBC 4.0 functions requires JDK 7 and usage of JDBC 4.1 functions requires JDK 8. If no Java development is going to be done, then the Linux system needs only a JRE to be installed. You can find more details about the JDBC installation at: https://ibm.biz/BdRzud.

The next consideration is modifying the CLASSPATH environment variable. The CLASSPATH needs to contain the JAR file that supports the JDBC driver and the DB2 Connect license file. Refer to the “Using the DB2 Connect middleware” section for more information about the DB2 Connect licensing requirements.

The license file name is db2jcc_license_cisuz.jar. The JDBC driver JAR file name depends on the level of required JDBC functionality. The db2jcc.jar file supports JDBC 3.0 and earlier functions while the db2jcc4.jar file supports JDBC 3.0 and earlier along with JDBC 4.0 and later levels. This example deployment requires only the JDBC 3.0 functionality, so the CLASSPATH is set in the following manner.

```
export CLASSPATH=.:/root/dsdriver/java/db2jcc.jar
export CLASSPATH=$CLASSPATH:/root/dsdriver/java/db2jcc_license_cisuz.jar
```

Obtaining a connection to DB2 for i

When Type 4 connectivity is used, the DB2 Universal Driver connects directly to the DB2 for i databases through TCP/IP. The Data Server Driver configuration file is not used. The code snippet in Example 15 shows how to use the DB2 Universal Driver to obtain a connection to DB2 for i.

```
try {
     Class.forName("com.ibm.db2.jcc.DB2Driver"); [1]
     } catch(java.lang.ClassNotFoundException e) {
       System.err.print("ClassNotFoundException: ");
       System.err.println(e.getMessage());
     }

try {
     DriverManager.getConnection(
                        "jdbc:db2://mysystem.mydomain.com:446/MYSYSTEM", [2]
                        "db2user", "userpwd");
     …
     } catch(SQLException ex) {
       System.err.println(
         "SQLException: " + ex.getMessage());
     }
```

Example 15: Code snippet that uses the DB2 Universal Driver to obtain a connection to DB2 for i

At **[1]**, an instance of the DB2 Universal JDBC Driver is initiated. Then at **[2]**, the driver is used to obtain a connection to the System i model. Notice the proper form of the database URL that is required by the driver.

Troubleshooting

The methods used for troubleshooting connections made with DB2 Connect middleware are similar to those discussed previously for IBM i middleware in the “DB2 engine feedback” section.

A key difference to remember is that the DRDA-based requests run in IBM i server jobs that are called QRWTSRVR. Note that a QRWTSRVR job is assigned to a client after the connection is established. Therefore, you need to set a breakpoint in the Linux application after the connection has been established before determining the IBM i server job that has been assigned.

Summary

This white paper has made you aware of the different interfaces that IBM provides for accessing DB2 for i data from applications running on Linux and the techniques for streamlining the data-access performance in these Linux applications.

Appendix A: Resources

The following websites provide useful references to supplement the information contained in this paper.

- IBM i Information Center
  http://publib.boulder.ibm.com/infocenter/iseries/v7r1m0/index.jsp

- IBM i on PartnerWorld
  ibm.com/partnerworld/i

- IBM i Access Client Solutions Linux Application Package
  ibm.com/systems/power/software/i/access/solutions.html

- Obtaining Access Client Solutions
  ibm.com/support/docview.wss?uid=nas8N1010355

- Linux ODBC Connection Keywords
  pic.dhe.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzaik%2Fconnectkeywords.htm

- DB2 for i SQL Performance Workshop
  ibm.com/systems/power/software/i/db2/education/performance.html

- IBM Redbooks: SQL Performance Diagnosis on IBM DB2 for i
  ibm.com/redbooks/abstracts/sg246654.html

- IBM i cwbtrc utility
  pic.dhe.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzatv%2Frzatvcwbtrc.htm

- Toolbox JDBC Driver installation instructions
  pic.dhe.ibm.com/infocenter/iseries/v7r1m0/index.jsp?topic=%2Frzahh%2Frzahnm09.htm

- IBM DB2 for Linux, UNIX and Windows Information Center
  pic.dhe.ibm.com/infocenter/db2luw/v10r5

- DB2 Connect Unlimited Edition for System i
  ibm.com/software/data/db2/db2connect/edition-uei.html

- IBM Data Server Driver Package download

ibm.com/support/docview.wss?rs=4020&uid=swg27016878

- IBM Data Server Driver Package Installation Instructions

  ibm.biz/BdRzv6

- IBM data server driver configuration file

  ibm.biz/BdRz8L

- DB2 Universal JDBC Driver Installation Instructions

  ibm.biz/BdRzud

Appendix B: Source listing of testJDBC.java

This is the source listing that is referenced in the “Techniques for improving JDBC performance” section.

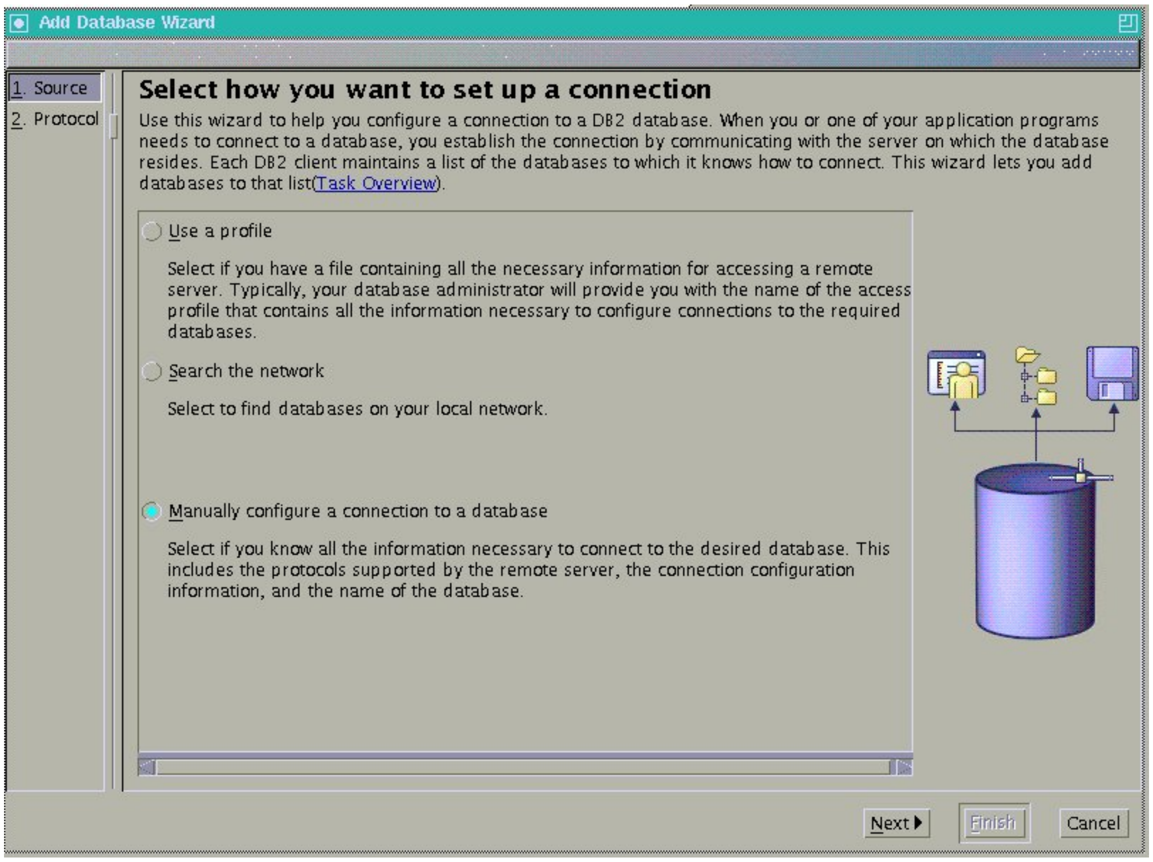
```
import java.sql.*;
import javax.naming.*;
import java.util.Hashtable;
import java.util.*;
import javax.sql.*;
import com.ibm.as400.access.*;

public class TestJDBC
{
 public static void main(java.lang.String[] args)
                   throws Exception
    {

     int numOfLoops = 5000;
     int outerNumOfLoops = 5;
     boolean firstExec = true;
     Connection con;
     Statement s;
     PreparedStatement ps;
     PreparedStatement pstmtdel;
     CallableStatement cstm;
     PreparedStatement pstmt;
     Hashtable<String,String> env = new Hashtable<String, String>();

     env.put(Context.INITIAL_CONTEXT_FACTORY,
     "com.sun.jndi.fscontext.RefFSContextFactory");
     Context ctx = new InitialContext(env);
     System.out.println("Deploying connection pooling data source");
     deployDataSource();
     // Do the work with connection pooling only.
     AS400JDBCConnectionPoolDataSource tkcpds =
(AS400JDBCConnectionPoolDataSource)ctx.lookup("jdbc/ToolkitDataSource");
     // Create an AS400JDBCConnectionPool object.
     AS400JDBCConnectionPool pool = new AS400JDBCConnectionPool(tkcpds);
     // Adds 1 connection to the pool that can be used by the
     //application (creates the physical database connections based on
     //the data source).
     pool.fill(1);
     System.out.println("\nStart timing Toolkit connection pooling...");
     long startTime = System.currentTimeMillis();
     for (int i = 0; i < outerNumOfLoops; i++) {
       con = pool.getConnection();
       con.setAutoCommit(false);
       // delete
       if ( i == 0)
        {

         long startDelete = System.currentTimeMillis();
         cstm =
            con.prepareCall(
             "CALL qsys.qcmdexc('CLRPFM FILE(DB2USER/COFFEES)'," +
             "0000000028.00000)");
         cstm.executeUpdate();
         cstm.close();
         long endDelete = System.currentTimeMillis();
         System.out.println("Delete all records : " +
                            (endDelete - startDelete) + "ms elapsed.");
        }
       String sqlinstr;
       long startInsert = System.currentTimeMillis();
       // insert
      pstmt =
        con.prepareStatement("INSERT INTO COFFEES VALUES(?, ?, ?, ?, ?)");
      for (int j = 0; j < numOfLoops; j++) {
        pstmt.setString(1, "Kona_" + Integer.toString(i));
        pstmt.setInt(2, 150);
        pstmt.setFloat(3, 9.99f);
        pstmt.setInt(4, 0);
        pstmt.setInt(5, 0);
        pstmt.addBatch();
       }
      int [] updateCounts = pstmt.executeBatch();
      con.commit();
      pstmt.close();
      long endInsert = System.currentTimeMillis();
      System.out.println("Time elapsed for " + numOfLoops +  " inserts "
                         + (endInsert - startInsert));
      // select
      long startSelect = System.currentTimeMillis();
      ps = con.prepareStatement("SELECT * FROM COFFEES");
      ResultSet rs = ps.executeQuery();
      rs.next();
      rs.close();
      con.commit();
      ps.close();
      long endSelect = System.currentTimeMillis();
      System.out.println("Time elapsed for select "  +
                         (endSelect - startSelect));
      con.setAutoCommit(true);
      con.close();
     }
    long endTime = System.currentTimeMillis();
    System.out.println("Time elapsed for " + outerNumOfLoops +
                       " loops  "  + (endTime - startTime));
    System.exit(0);
    }

    private static void deployDataSource()
    throws Exception
    {
     String serverName = "teraplex" ;
     String databaseName = "TERAPLEX" ;
     String userName = "db2user" ;
```

```
         String password = "db2pwd" ;

        // Establish a JNDI context and bind the connection pool data source
         Hashtable<String, String> env = new Hashtable<String, String>();
         env.put(Context.INITIAL_CONTEXT_FACTORY,
                 "com.sun.jndi.fscontext.RefFSContextFactory");
         Context ctx = new InitialContext(env);
         // Create an AS400JDBCConnectionPool object.
         AS400JDBCConnectionPoolDataSource tkcpds = new
                     AS400JDBCConnectionPoolDataSource();
         tkcpds.setServerName(serverName) ;
         tkcpds.setDatabaseName(databaseName) ;
         tkcpds.setUser(userName) ;
         tkcpds.setPassword(password) ;
         tkcpds.setDescription("Toolkit Connection Pooling DataSource");
         tkcpds.setSavePasswordWhenSerialized(true);
         tkcpds.setExtendedDynamic(true);
         tkcpds.setPackage("JDBCDS");
         tkcpds.setPackageCache(true);
         tkcpds.setPackageCriteria("select");
         //enable the following properties as required
         //cpds.setTrace(true);
         //tkcpds.setServerTraceCategories(
         //       AS400JDBCDataSource.SERVER_TRACE_DEBUG_SERVER_JOB);

         ctx.rebind("jdbc/ToolkitDataSource", tkcpds);

        }
    }
```

Appendix C: Client Configuration Assistant steps

The following steps show to use the Client Configuration Assistant through the process of configuring a connection to a DB2 for i server. The Client Configuration Assistant is currently provided only with the earlier versions of DB2 Connect (releases prior to version 10).

1. Log in to the Linux partition as the DB2 administrator.
2. Start the Client Configuration Assistant (db2ca from the command prompt). It is assumed that VNC or a X-Windows server is used on the workstation so that the db2ca GUI can be properly rendered.
3. From the **Selected** drop-down list, select **Add Database using Wizard**. The **Select how you want to setup a connection** page appears. Select **Manually configure a connection to a database** and click **Next** (refer to Figure 11).



Select how you want to set up a connection

Use this wizard to help you configure a connection to a DB2 database. When you or one of your application programs needs to connect to a database, you establish the connection by communicating with the server on which the database resides. Each DB2 client maintains a list of the databases to which it knows how to connect. This wizard lets you add databases to that list(Task Overview).

Use a profile

Select if you have a file containing all the necessary information for accessing a remote server. Typically, your database administrator will provide you with the name of the access profile that contains all the information necessary to configure connections to the required databases.

Search the network

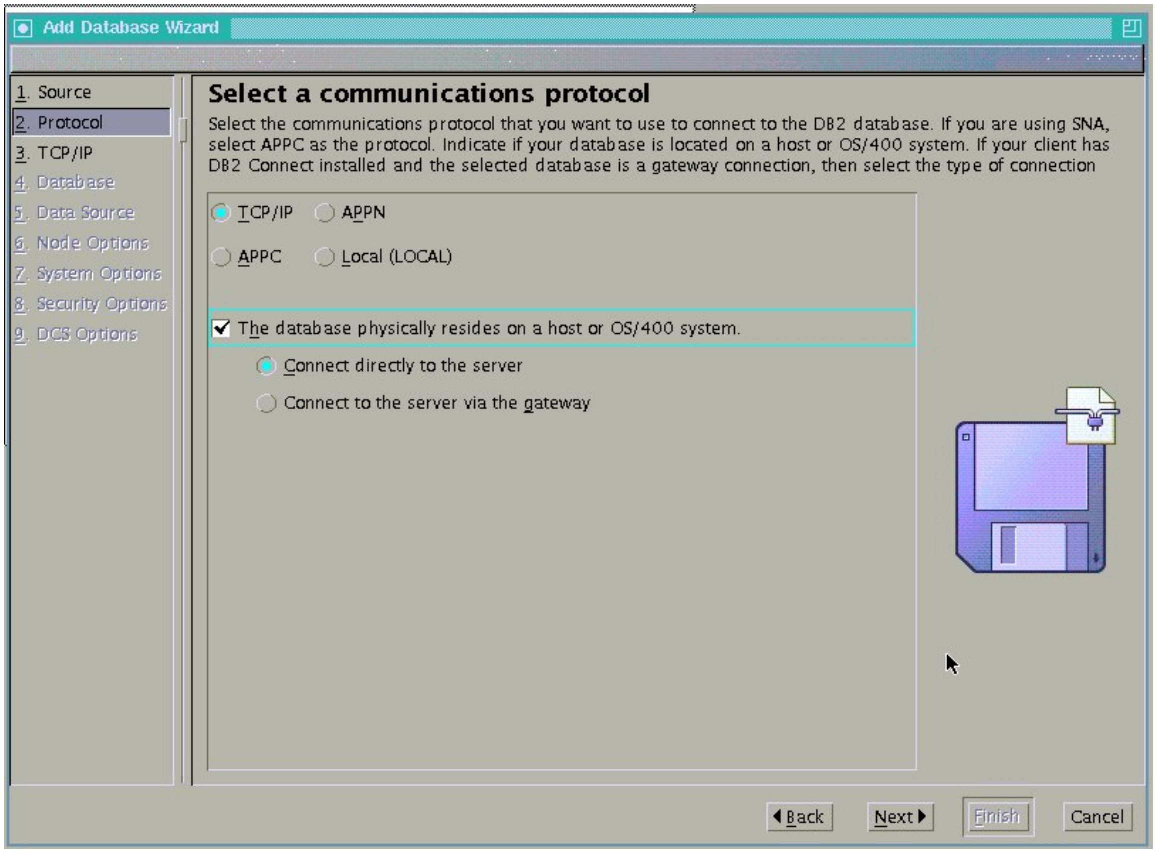
Select to find databases on your local network.

[✓ Manually configure a connection to a database]

Select if you know all the information necessary to connect to the desired database. This includes the protocols supported by the remote server, the connection configuration information, and the name of the database.

Figure 11: Selecting manual configuration

4. On the **Select a communications protocol** page, select **TCP/IP** and select the **The database physical resides on a host or OS/400 system** check box. Then, select **Connect directly to the server** and click **Next**.



Select a communications protocol

Select the communications protocol that you want to use to connect to the DB2 database. If you are using SNA, select APPC as the protocol. Indicate if your database is located on a host or OS/400 system. If your client has DB2 Connect installed and the selected database is a gateway connection, then select the type of connection

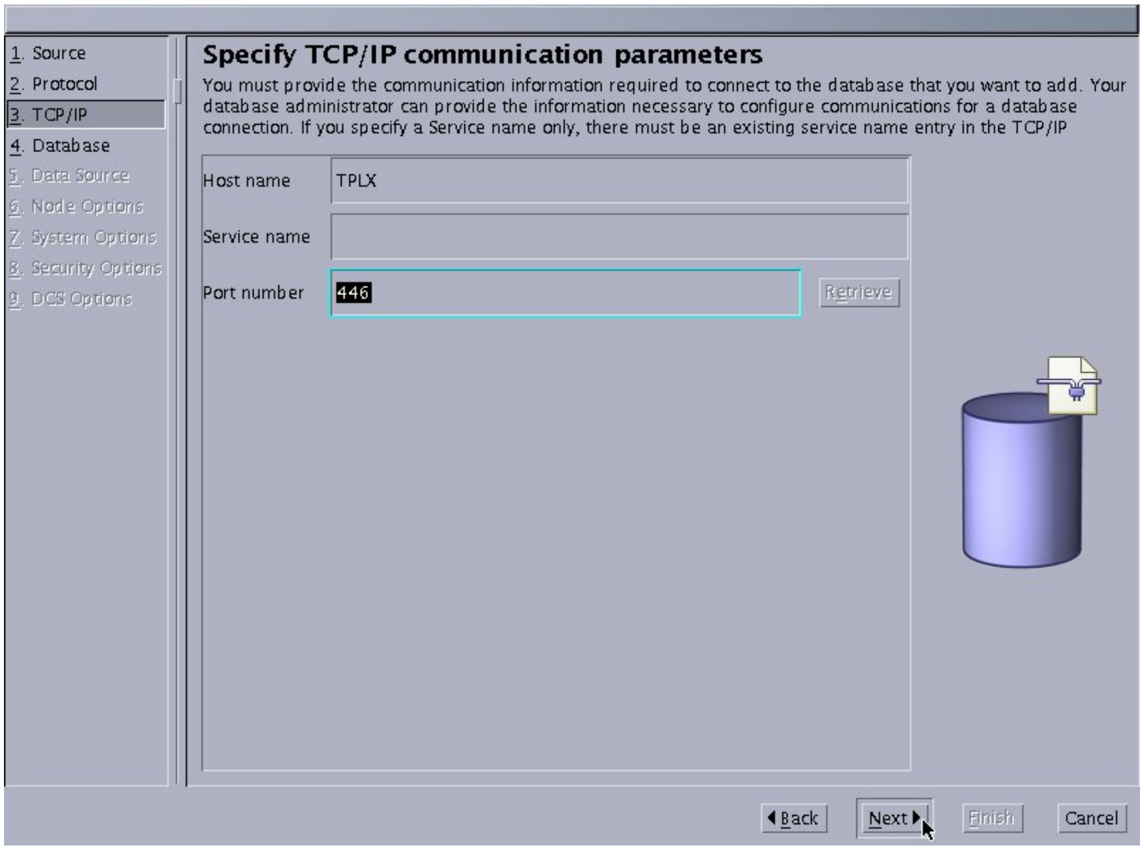
[✓ TCP/IP] APPN
APPC Local (LOCAL)

[✓ The database physically resides on a host or OS/400 system.]

[✓ Connect directly to the server]
Connect to the server via the gateway

Figure 12: Selecting a communications protocol

5. On the **TCP/IP** tab, enter the host name of the IBM i system. The port number must be 446 (refer to Figure 13). Click **Next.**



Specify TCP/IP communication parameters

You must provide the communication information required to connect to the database that you want to add. Your database administrator can provide the information necessary to configure communications for a database connection. If you specify a Service name only, there must be an existing service name entry in the TCP/IP

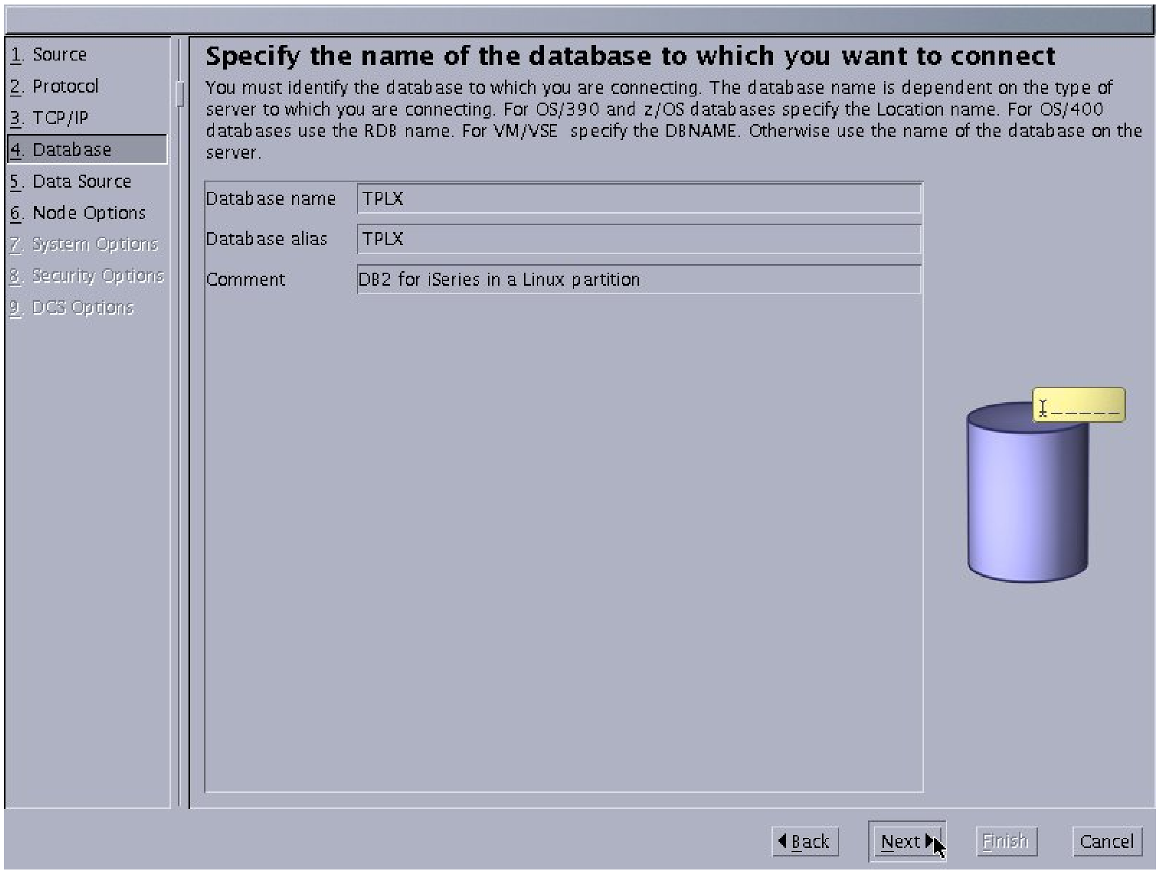
Host name: TPLX

Service name:

Port number: 446

Figure 13: Specifying TCP/IP communication parameters

6. On the **Database** tab, enter the relational database name. Use the RDB entry name (specified in step 2 in the “Configuring the IBM i” section. Click **Next**.



Specify the name of the database to which you want to connect

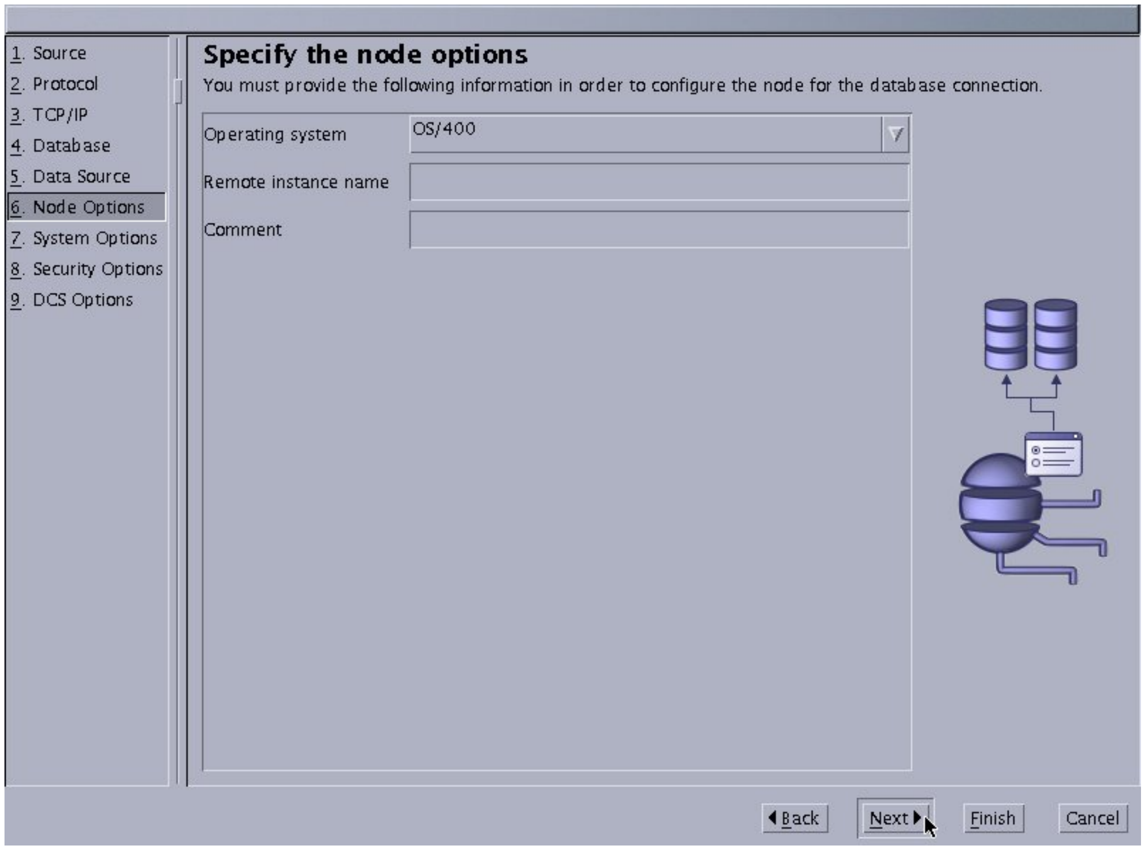
You must identify the database to which you are connecting. The database name is dependent on the type of server to which you are connecting. For OS/390 and z/OS databases specify the Location name. For OS/400 databases use the RDB name. For VM/VSE specify the DBNAME. Otherwise use the name of the database on the server.

Database name	TPLX
Database alias	TPLX
Comment	DB2 for iSeries in a Linux partition

Figure 14: Specifying the remote database

7. If you plan to use the ODBC applications, select **Register this database for ODBC as a system data source** on the ODBC tab. Click **Next**.

8. On the **Specify the node option** page, select **OS/400** from the Operating System drop-down list and click **Finish**.



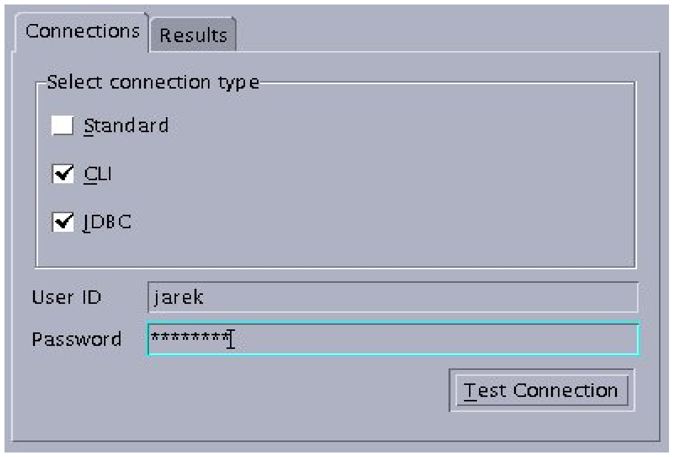
Specify the node options

You must provide the following information in order to configure the node for the database connection.

Operating system: OS/400
Remote instance name:
Comment:

Figure 15: Specifying the node options

9. The Test Connection dialog appears, allowing you to verify that the connection works. Select the **CLI** and **JDBC** check boxes. You are also prompted for an IBM i user ID and password (refer to Figure 16).

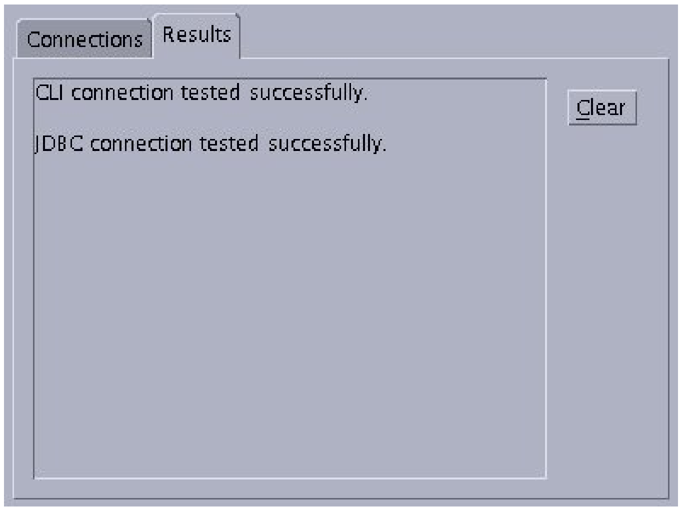


Select connection type
- [ ] Standard
- [✓] CLI
- [✓] JDBC

User ID: jarek
Password: ********

Figure 16: Testing the connection

10. The successful connection to the database server is confirmed by the message box shown in Figure 17.

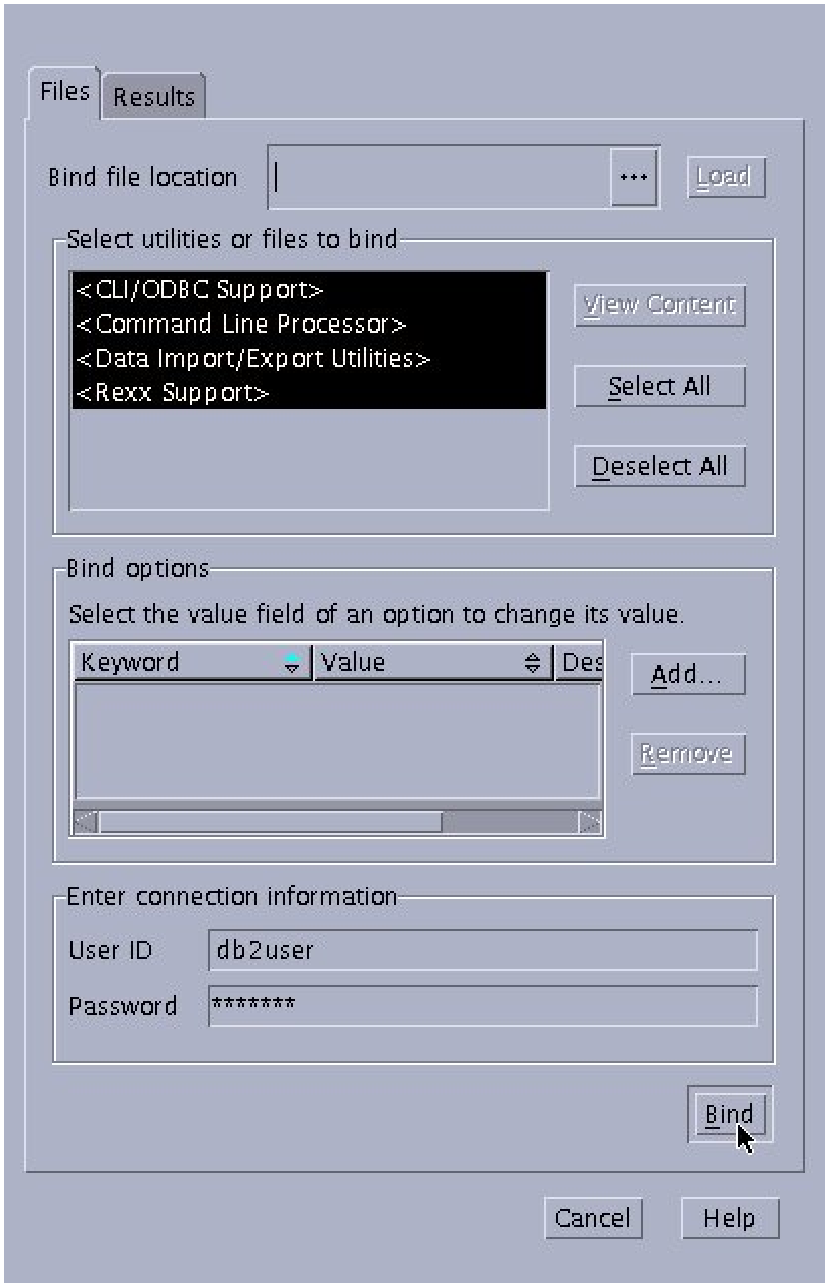


```
CLI connection tested successfully.

JDBC connection tested successfully.
```

Figure 17: Test connection results

11. A newly configured database connection appears in the main DB2 Configuration Assistant window. It is recommended that the DB2 utilities are also bound to the IBM i database. The bind process creates the necessary SQL packages on the remote database. In the main Configuration Assistant window, click the **TPLX** database entry and then click **Selected → Bind**…

12. In the Bind dialog box, click **Select All** and then enter the connection information (refer to Figure 18). Click **Bind**. The Results message box appears. It allows you to monitor the progress of the binding process. Watch for any error and warning messages.



<CLI/ODBC Support>
<Command Line Processor>
<Data Import/Export Utilities>
<Rexx Support>

Select the value field of an option to change its value.

User ID db2user

Password *******

Figure 18: Binding the DB2 utilities

This concludes the Client Configuration Assistant steps.

About the authors

Kent Milligan is a Senior Certified DB2 for i Consultant on the ISV Enablement team for IBM i in Rochester, Minnesota. After graduating from the University of Iowa, Kent spent the first eight years of his IBM career as a member of the AS/400 and DB2 development group in Rochester. He speaks and writes regularly about relational database topics and DB2 for i. You can reach Kent at kmill@us.ibm.com.

Jarek Miszczyk is a Senior Software Engineer in IBM Software Defined Systems Development organization. Before joining the Software Defined Systems organization, Jarek provided DB2 for i expertise while working for the ISV Enablement and International Technical Support Organization (ITSO) teams in the IBM Rochester lab.

@IBMSystemsISVs

Trademarks and special notices