MAKING ORACLE AND SQLJ WORK FOR YOU

John Jay King, King Training Resources

Oracle and Java are an uncommonly good pairing; Oracle provides relational database for most environments and Java provides code that works in most environments. As in many areas, Oracle was ahead of the curve and made a commitment to Java long before it became fashionable. Oracle was the first database to incorporate a Java Virtual Machine (JVM) into the database and Java Stored Procedures/Functions have become part of the main stream. Java's ability to use Oracle's database is provided by means of JDBC (Java DataBase Connectivity) and SQLJ. JDBC was built upon the ODBC standard and allows Java programs to access the commonly used relational database products including Oracle. SQLJ provides a less Java-like interface, with statements similar to embedded SQL in other languages. This paper addresses SQLJ.

SQLJ provides a programmer-friendly mechanism to query and update database tables and execute stored procedures. Oracle supports ISO-standard SQLJ and provides customizations as well. Java programs access the database via "driver" programs, connecting to the database and manipulating data as needed. Typically driver and connection code is tailored to a vendor's database. Industry-standard SQLJ may be used to make code fairly portable, but, vendor-specific SQL variations are also available. Oracle's JDBC drivers and SQLJ mechanism provides a mechanism that is tailored for the Oracle database and its data. Oracle SQLJ also allows programmers to tune the connection to the database in a way that is not possible using generic code. Developers must decide between the need for portability and the need for maximum performance. Where portability between database products is a major concern the "generic" SQLJ should be used. If performance is the most important issue, then, Oracle's SQLJ mechanism should be used. This article will (in a few pages…) introduce the Java-SQLJ syntax necessary to query and update Oracle table data as well as to execute Oracle stored procedures and functions. Along the way some of the more-important Oracle extensions will be explored and explained. The article assumes that you are familiar with Oracle SQL and have some basic knowledge of Java.

Oracle and Java

Oracle supports Java in a variety of ways. Oracle9i (and Oracle8i) provides a Java Virtual Machine (JVM) built in to the database. Oracle was the first major vendor to include a JVM in the database. Oracle provides JDBC drivers to allow Java to use the database in different ways depending upon the need. Oracle provides an SQLJ translator allowing the use of embedded SQL in Java. JDeveloper is an Integrated Development Environment (IDE) Oracle provides that may be used to create Java programs using JDBC or SQLJ. Oracle's database support includes CORBA compliance along with support for Java Beans and Enterprise Java Beans (EJBs). Finally, Oracle's Application Server is a capable web server providing support for Java, servlets, JSPs (Java Server Pages), and EJBs (Enterprise Java Beans).

SQLJ and Java

SQLJ source code combines Java code with special SQLJ statements in a file with a .sqlj suffix. The SQLJ source code (.sqlj file) is input to an SQLJ translator that checks the syntax of SQLJ statements, and, if all is well converts the SQLJ code into a standard Java source file (.java suffix) that may be compiled, and sometimes a file that contains "serialized profiles" that match the program (.ser suffix). The translated Java code is then compiled in the normal fashion. Translator option "codegen=iso" causes generic code to be used and creates a serialized profile (.ser) output file. Using the default translator option of "codegen"=oracle causes Oracle-specific code to be generated and a serialized profile is not generated. Oracle 9i Stored Procedures and Stored Functions must be created as Oracle-specific. SQLJ uses JDBC (Java Data Base Connectivity) drivers to access the database from Java.

Static vs. Dynamic SQL

One advantage of using SQLJ is that SQL statements are parsed and optimized at the time of SQLJ compilation, reducing the amount of work done at execution time. This "static" SQL is also easier to tune than dynamic SQL.

Database Connectivity via JDBC

Sun provides JDBC (Java Data Base Connectivity) as a tool to access SQL databases from Java. Oracle's JDBC drivers support JDBC 1.22 fully, most of JDBC 2.0, and with Oracle9i R2 support for most of JDBC 3.0 (the current release). SQLJ programs use of JDBC drivers allows programmers to use SQL to: Connect to a database, Query a database, Update a database, and execute Database Stored Procedures/Functions (written in PL/SQL or Java).

JDBC programs are database vendor and platform independent (mostly) unless Oracle-specific features are used. SQLJ programs use

JDBC Drivers

Oracle supports four different JDBC drivers: the Oracle Thin JDBC driver, Oracle OCI JDBC Driver, Oracle Server Internal JDBC driver, and the Oracle Server Thin JDBC Driver.

The Oracle Thin JDBC driver does not require an Oracle client installation, it creates its own Oracle Net (Net 8/Sql*Net) connection. While Oracle’s documentation describes the Thin JDBC driver as downloadable, it is approximately 900K. The Thin JDBC driver should be used for Applets and Applications running in a client environment where an Oracle Client is not installed. Servlets, JSPs, and other web server code might also use the Thin JDBC driver when an Oracle Client is not practical. The Thin JDBC driver assumes a TCP/IP connection.

The Oracle OCI JDBC driver requires a local Oracle Client installation and uses its existing Oracle Net (Net 8 / Sql*Net) connection. The OCI Client is good for Servlets, JSPs, Applications, and beans executing in a web server environment. The cost of the Oracle Client installation will probably be offset by flexibility provided by the OCI Client interface.

Oracle's documentation indicates that the OCI JDBC driver outperforms the Thin client JDBC driver. Contrary to the Oracle documentation, some users have found that occasionally the Thin client JDBC driver performs as well or better than the OCI client. As always with performance your results may vary, there is no substitute for thorough testing.

The Oracle Server Internal JDBC driver (sometimes known as the KPRB driver) is used by the Oracle database to support Java stored procedures (also functions and packages) and EJBs. This driver supports communications between PL/SQL and Java. The Internal JDBC driver supports Java 1.2.x and does not incorporate all of the newest features.

An Oracle Server Thin JDBC driver is also available, it is intended for Java programs running in one Oracle database that want to connect with some other Oracle database.

Both the Thin JDBC and OCI JDBC drivers work with Oracle9i and Oracle8i, objects are not supported for Oracle 8, and neither objects nor LOBs are supported for Oracle 7.x. The Oracle Server JDBC drivers became available in Oracle 8.1.5 but can access data in Oracle 8.1.4.

The actual driver files are stored as zip files located in the ORACLE_HOME JDBC directories and have the names classes111.zip (Java 1.1.x) and classes12.zip (Java 1.2, 1.3, and 1.4). Careful! Oracle driver zip files use the same names from version to version and the only way to tell them apart is the size and date of the file's creation. Like most things in Oracle the .zip files are downward but not upward compatible. Oracle9i also introduces the ojdbc14.jar that includes only classes specific to Oracle9i and JDK 1.4. One (and only one) of these files must be present in the Java CLASSPATH environment variable. Do not "unzip" these files, Java will use them in their zipped form.

In addition to the JDBC driver files, SQLJ requires that you include two other files in the CLASSPATH: runtime12ee.zip (Oracle9i EE) or runtime12.zip (Oracle9i) or runtime111.zip (Oracle8/8i), and translator.jar.

Using SQLJ

The steps for using SQLJ depend upon whether you want to mix JDBC code with the SQLJ or not. If using SQLJ only the "Oracle" class is used to connect to the database, then SQLJ allows SQL (or PL/SQL) to be executed. If JDBC objects used the steps are generally as follows: load or register the JDBC driver, connect to the database using JDBC, create a statement object, execute an SQL statement and process the results, finally close the result set and connection.

Compiling and Executing

The SQLJ source code file (.sqlj) is compiled from the command line or from within an IDE, the Java program created (.java) is then compiled and creates a useable bytecode (.class) file:

```
sqlj OraSQLJSelect.sqlj
```

Using SQLJ to Connect to the Database

Using pure SQLJ means using "Oracle" class static methods including connect:

```
Oracle.connect(MyClass,connect.properties);
```

The connect method both registers the driver and opens the database connection using a connect.properties file.

```
# Users should uncomment one of the following URLs or add their own.
# (If using Thin, edit as appropriate.)
sqlj.url=jdbc:oracle:thin:@myserver:1521:mysid
#sqlj.url=jdbc:oracle:thin:@localhost:1521:ORCL
#sqlj.url=jdbc:oracle:oci8:@
#sqlj.url=jdbc:oracle:oci7:@
# User name and password here (edit to use different user/password)
sqlj.user=scott
sqlj.password=tiger
```

Using JDBC to Load or Register the jdbc driver and Connect to the Database

Instead of Oracle.connect you might load and register a JDBC driver manually. This is one of the two parts of JDBC that is most impacted by switching database vendors. Before you go any farther, make sure that the JDBC driver has been made available to the JVM (in the classpath). Also, you might want to make sure that the most current driver is used. Use one of these two mechanisms:

```
DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());
```

or

```
Class.forName("oracle.jdbc.driver.OracleDriver()");
```

Some web servers pre-register JDBC drivers, so, this step might be omitted by Servlet or JSP code.

Connections are the other part of JDBC that is database-vendor dependent. Oracle’s connection string varies depending upon the JDBC driver in use. A DBA's (DataBase Administrator's) help may be required to properly format the connect string and identify the appropriate settings. Note: It is a bad idea to hard-code user id and password information in code as shown in these examples, be sure to prompt the current user for the necessary information. When finished, be sure to close the connection, the connection close is often placed in a *finally* block.

Thin client connection:

```
Connection conn =
      DriverManager.getConnection(jdbc:oracle:thin:@host:port:sid,uID,uPswd);
```

@host	Host computer DNS name or TCP/IP address
port	TCP/IP port number host Oracle is listening on (1521 is the default)
sid	Oracle System Identifier
uID	Oracle Userid
uPswd	Oracle User Password

OCI client connection:

```
Connection conn =
      DriverManager.getConnection(jdbc:oracle:oci:@sid,uID,uPswd);
```

oci	oci=Oracle9i, oci8=Oracle8i/8, oci7=Oracle 7
sid	Oracle System Identifier or Net Service Name
uID	Oracle Userid
uPswd	Oracle User Password

Internal database connection:

```
Connection conn =
      DriverManager.getConnection(jdbc:default:connection);.
```

The connections depicted above represent a single connection for each program. For performance reasons your DBAs might set up "Connection Pools" to reduce the impact of multiple SQLJ or JDBC users.

It is the developer's responsibility to make sure that every connection is closed properly before a program ends (normally or abnormally). This is a good use for Java's *finally* block.

Connection pools

Creating and destroying database connections is one of the most expensive operations SQLJ or JDBC performs. Performance may sometimes be improved dramatically by using “Connection Pools.” Connection Pools represent a set of existing database connections that application programs share, eliminating the cost required to create/destroy a connection each time a program uses the database. Creating and managing Connection Pools is often the province of the DBA staff, though, applications may create them manually.

Caching Connection objects provides significant performance improvements. Using Connection Pools takes advantage of JDBC 2.0's JNDI (Java Naming and Directory Interface) technology using ConnectionPoolDataSource, DataSource, and PooledConnection interfaces.

Once again, the Oracle class provides a static method to help:

```
   DefaultContext ctx =
       Oracle.getConnection("jdbc:oracle:thin:@host:port:sid,uID,uPswd");
```

The contract (ctx above) may then be used by SQLJ statements. Note: Avoiding the DriverManager and using contexts allow use of shared connection pools.

It is still important that the developer make sure that every connection is closed properly before a program ends (normally or abnormally). Closing Connection Pool connections makes a connection available to another user, it does not actually close the connection. Again, this is a good use for Java's *finally* block.

Create a Statement Object

The Statement object is a container for the SQL statement to be executed, it is not the SQL statement. For maximum performance and flexibility, it might be necessary to use OracleStatement objects. Be sure to close the Statement object when finished, this is code often found in a *finally* block.

Standard JDBC Statement:

```
   Statement stmt = conn.createStatement();
```

Oracle-Specific Statement:

```
   OracleStatement stmt = conn.createStatement();
```

SQLJ Statements

SQLJ statements begin with the string "#sql" and have two basic varieties, declarations and executable statements. SQLJ declarations follow the "#sql" with the declaration of a class. Declarations allow creation of one of two kinds of objects: iterators and connection contexts.

Iterators are very similar to PL/SQL cursors and JDBC result sets and allow processing of SQL query results one row at a time. Connection contexts are used to support specific operations against Oracle database objects, targeted for environment where multiple database connections might be in use (not discussed further in this paper).

An iterator defines the variables (Java datatypes and names) that represent one row of a query's output. The iterator is then used to receive the result of a query. Once a query has loaded an iterator a Java program may then process the rows one at time very much like a PL/SQL program processes a cursor.

Multiple Row Query

First, define the iterator data type and the variables that represent one output row:

```
    #sql iterator MyIter (String ename, String job);
```

Next, create a variable of the iterator type created above, execute an SQL query, and use the iterator object to hold the results:

```
    MyIter iter;
    #sql iter = { select ename,job from emp };
```

Finally, using a loop process the results of the query one row at at time:

```
       // Loop through results set and print the employee info
       while (iter.next ())
         System.out.println ("Employee: " + iter.ename()
                         + " is a " + iter.job());
```

It is a good idea to issue the Oracle.close() in a Java "finally" block.

```
/*
 * SQLJ application using Oracle "thin" driver
 *
 * @version 1.01 04/01/2000
 * @author  John Jay King
 */

import java.sql.*;        // need for JDBC
import oracle.sqlj.runtime.Oracle; // need for sqlj
import oracle.sqlj.runtime.Oracle.*;

/** Sample use of Oracle "thin" JDBC connection */
class OraThinSQLJ {
  // sqlj iterator for SELECT
  #sql iterator MyIter (String ename, String job);
  public OraThinSQLJ() {
 try {
     // Load the Oracle JDBC driver
     DriverManager.registerDriver(new oracle.jdbc.driver.OracleDriver());

     // Connect to the database, connection string format:
     //  jdbc:oracle:thin:@**hostcomputer**:**port**:**oraclesid**

     // Use connect.properties file to connect
     Oracle.connect(OraThinSQLJ.class,"connect.properties");

     MyIter iter;
     #sql iter = { select ename,job from emp };

     // Loop through results set and print the employee info
     while (iter.next ())
       System.out.println ("Employee: " + iter.ename()
                         + " is a " + iter.job());
 } // end try
 catch (SQLException e) {
     System.out.println("SQL error: SQLCODE = " + e.getErrorCode());
     e.printStackTrace();
 } // end catch
 finally {
     try {
         // Close
          Oracle.close();
      }
      catch (SQLException e) {
          System.out.println("Error closing connection");
      }
 } // end finally

  } // end constructor
} // end OraThinSQLJ class
public class OraSQLJSelect {
  public static void main (String args [])
       throws SQLException {
      OraThinSQLJ myClass = new OraThinSQLJ();
 } // end main
} // end OraSQLJSelect class
```

Host Variables

Single SQL statements may use host variables. Host variables are Java program variables that may be used as input or output from the SQL statement. Variables may be defined concerning their use as IN, OUT, or INOUT (like PL/SQL but not the same). Host variables are preceded by a colon (:).

Single-row query:

Query using host variables without usage indicators:

```
#sql { SELECT ename,sal
           INTO :empName, :empSal
           FROM emp
           WHERE empno = :empNo };
```

Query using host variables with usage indicators:

```
#sql { SELECT ename,sal
           INTO :OUT empName, :OUT empSal
           FROM emp
           WHERE empno = :IN empNo };
```

To execute an update/insert/delete:

```
        #sql { update emp
               set sal = :in empSal
               where empno = :IN empNo };
        #sql { rollback }; // or commit
```

AutoCommit

Using SQLJ's Oracle.connect turns off the normal JDBC autocommit, if using the JDBC Connection object a program must manually turn off autocommit.

When performing JDBC access it is important to understand that the default JDBC mechanism includes an autocommit feature. That is, JDBC issues a Commit after each statement execution. Autocommit is often not the best performance choice. JDBC programmers may "shut off" the autocommit programmatically.

```
   conn.setAutoCommit(false);
```

Once autocommit has been "shut off" it becomes necessary to manually issue Commit and Rollback statements manually as well:

```
    conn.commit();
```

and

```
   conn.rollback();
```

Batched Update/Insert/Delete

Sometimes a program will be doing a large number of Insert, Update, or Delete statements requiring many network travels to accomplish the work. *Update batching* is provided by SQLJ to allow Insert/Update/Delete statements to be batched. Batched statements are usually part of some kind of loop. Batched statements are held in memory and submitted to the database together (as a batch).

Using Stored Procedures

Using Stored Procedures is a simply a matter of knowing the procedure/function name, parameters, and return type (for functions). The example below use host variables as discussed earlier.

To execute a Stored Procedure:

```
        #sql { begin addem(:IN inVal1, :IN inVal2, :out outVal); end; };
```

JDBC Cleanup

It is a good idea to put the Connection close statement in a *finally* block:

```
   finally {
    try {
        // Close connection or connection-pool connection
        Oracle.close();
    }
    catch (SQLException e) {
        // process error closing connection
   }
     } // end finally
```

Performance Tips

Here are a few ideas repeated from earlier in this paper that might impact performace. As always with performance there is no substitute for repeated testing in the actual execution environment:

- SQLJ's Static SQL is more efficient for repetitive executions,
- SQLJ's Static SQL is more easily tuned,

- JDBC's autocommit (the default unless using Oracle.connect) may be a poor performer when many SQL statements are involved, controlling commit/rollback manually will probably improve performance,
- If many Insert/Update/Delete operations are involved, Batched SQL might improve performance more,
- If many Insert/Update/Delete operations are involved, a Stored Procedure might work best.

Oracle Documentation

- Thorough documentation may be found for Oracle's SQLJ Developer's Guide and Reference. A textbook is also available named "Oracle SQLJ Programming" from the Oracle press.
- Complete javadoc for Oracle JDBC may be found in the ORACLE_HOME/sqlj/doc directory of any Oracle8i or Oracle9i installation.
- Complete javadoc for the Oracle JDBC classes may be found in the ORACLE_HOME/jdbc/doc directory of any Oracle8i or Oracle9i installation.

Lots of papers and examples are also available at `http://technet.oracle.com/`.

Wrapping it all Up

SQLJ allows full use of Oracle from Java programs. SQLJ is simpler to code than JDBC, but, since it does not "look" like standard Java some Java programmers resist it. SQLJ looks and feels more like one of the Pro* pre-compiled languages or PL/SQL and is often an easy transition for experienced Oracle programmers. Oracle-specific features may be used to improve performance at the cost of portability.

About the Author

John King is a Partner in King Training Resources, a firm providing instructor-led training since 1988 across the United States and Internationally. John specialized in application development software on a variety of platforms including Unix, Linux, IBM mainframe, personal computers, and the web. John has worked with Oracle products and the database since Version 4 and has been providing training to Oracle application developers since Oracle Version 5. John develops and presents customized courses in a variety of topics including Oracle, DB2, UDB, Java, XML, C++, and various programming languages. He has presented papers at various industry events including: IOUG-A Live!, UKOUG Conference, EOUG Conference, RMOUG Training Days, MAOP-AOTC, NYOUG, and the ODTUG conference.

John Jay King
King Training Resources
6341 South Williams Street
Littleton, CO 80121-2627
U.S.A.
Phone: 1.303.798.5727 1.800.252.0652 (within the U.S.)
Fax: 1.303.730.8542
Email: john@kingtraining.com
Website: http://www.kingtraining.com