SQLJ — Part 1: SQL Routines using the Java™ Programming Language

Andrew Eisenberg
Progress Software Corp., Bedford, MA 01730
andrew.eisenberg@progress.com

Jim Melton
Oracle Corp., Sandy, UT 84093
jim.melton@acm.org

Introduction

The SQLJ informal group of companies has continued to work productively since last we wrote about them in December 1998 [1]. At that time we discussed SQLJ Part 0 [2], which had just been approved as NCITS standard SQL Part 10: Object-Language Bindings (SQL/OLB). This standard allows SQL statements to be embedded in the Java™ Programming Language.

In the first half of 1999, the SQLJ group (consisting of Cloudscape, Compaq (Tandem), IBM, Informix, Oracle, Sun, and Sybase) completed its work on the SQLJ Part 1 specification and requested that it be adopted by NCITS under its Fast Track process. In Sept. 1999 SQLJ Part 1 was adopted as NCITS 331.1-1999 [3] and it is now available for purchase from NCITS. It is worth mentioning that this specification is extremely approachable, with a lengthy tutorial section that introduces its more normative elements.

Sybase brought SQLJ Part 1 to the SQLJ group in early 1997. Phil Shaw, of Sybase, has acted as editor of this document throughout its development. SQLJ Part 1 allows Java classes, contained in Jar files, to be brought into a DBMS. Methods in these classes may then be used as the implementation of SQL stored procedures and stored functions (together referred to as stored routines).

Given how these methods are used, we'll provide a brief introduction to SQL routines before we discuss the features of SQLJ Part 1.

SQL Routines

SQL Part 4: Persistent Stored Modules (SQL/PSM) [4] introduced SQL routines in 1996. We will discuss them in this section, largely by example.

SQL functions have only input parameters, and produce a result of some designated SQL data type. Once a function is defined, it can be invoked from within any expression.

```
CREATE FUNCTION pub_year (INTEGER volume)
      RETURNS CHAR (4)
   BEGIN
      RETURN
         CAST (volume + 1971 AS CHAR(4));
   END ;
```

```
SELECT   *
FROM     sigmod_articles sa
WHERE    pub_year (sa.volume) = '1995';
```

In this example, the `pub_year` function is defined to take an integer argument, and returns a 4 character string. This function is then used in the `WHERE` clause of a query.

SQL procedures can have parameters that have a mode of either `IN`, `OUT`, or `INOUT` (`IN` is the default). They do not return values in the way that SQL functions do.

```
CREATE PROCEDURE pub_info
                    ( IN INTEGER volume,
                      IN INTEGER no,
                      OUT CHAR(4) yyyy,
                      OUT CHAR(3) mmm)
   BEGIN
      yyyy = pub_year (volume);
      CASE no
         WHEN 1 THEN SET mmm = 'MAR';
         WHEN 2 THEN SET mmm = 'JUN';
         WHEN 3 THEN SET mmm = 'SEP';
         WHEN 4 THEN SET mmm = 'DEC';
         ELSE SET mmm = '???';
      END CASE ;
   END
```

```
DECLARE p_year CHAR (4);
DECLARE p_month CHAR (3);
CALL pub_info (22, 3, p_year, p_month);
```

In this example, the `pub_info` procedure is defined to accept input parameters for volume and number, and then produces output values for the year and month that correspond to those numbers. After `pub_info` has been called, the variables `p_year` and `p_month` will contain `'1993'` and `'SEP'`, respectively. In both types of SQL routines, the body

---

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

consists of an SQL statement, which may be a compound (BEGIN/END) statement.

Both types of routines also can be defined as *external routines*. These routines have the same signature information, but instead of a body with SQL statements, they contain the name of an entry point in code written in some 3GL.

```
CREATE FUNCTION pub_year (INTEGER volume)
      RETURNS CHAR (4)
LANGUAGE c
EXTERNAL NAME 'pubyear@sigmod.dll'
PARAMETER STYLE GENERAL
```

The format of the string that identifies the external entry point is determined by the DBMS (sometimes influenced by the particular OS). The invocation of these external routines is no different than that of SQL routines.

Procedures have one final capability that we'll mention. They can return one or more result sets, along with the scalar values they return via their OUT and INOUT parameters. Call-level interfaces, such as ODBC and JDBC, have methods to allow an application to examine these result sets.

```
CREATE PROCEDURE article_count
                     (IN INTEGER a_volume)
     READS SQL DATA
     DYNAMIC RESULT SETS 1
   BEGIN
     DECLARE result CURSOR WITH RETURN FOR
       SELECT   no, COUNT(*)
       FROM     sigmod_articles sa
       WHERE    sa.volume = a_volume
       GROUP BY no ;

   OPEN c1;

   END ;
```

This example first states that it will return at most one result set. The procedure then declares and opens a cursor that, because it has been declared as WITH RETURN and because it has been left open, will return its result rows to the procedure's invoker as a result set.

Now that we've refreshed ourselves on SQL/PSM, we can continue on to the new material. Further information on SQL routines can be found in an earlier article of Andrew's [5].

Installing Jar Files

The first step that you must take in using SQLJ Part 1 is to import your Java classes into the database. Your classes must be in one or more Jar files. This is easy enough to do, using the jar utility that Sun provides with their Java SDK's.

Let us say that we have a Java class that contains an implementation of the pub_year SQL function we saw earlier.

```
public class Sigmod {

  public static String pubYear(int volume) {
    return String.valueOf (volume + 1971);
  }

}
```

Let us further say that this class has been compiled and placed in the Sigmod.jar file. We would install this Jar file in the database with the following statement:

```
CALL sqlj.install_jar
            ('file:///d:/Sigmod.jar',
             'sigmod_jar',
             0);
```

This operation will copy the contents of the Jar file (identified by a URL) into the database, and give an SQL name to the Jar file. The first two parts of the SQL name of the Jar file (catalog name and schema name) are not provided in this example, and so are implicitly defined (SQL has rules for this that it applies to the names of many types of database objects). Although it has the form of a procedure call, this statement is really acting as a DDL statement. The final argument of 0 indicates that a deployment descriptor is not to be used (we'll discuss deployment descriptors in a little while).

In the interest of brevity, we will simply mention that SQLJ Part 1 provides an sqlj.replace_jar procedure and an sqlj.remove_jar procedure that allow installed Jar files to be maintained and enhanced over time.

Creating Java Routines

For the contents of the Jar file to be useful, a Java Virtual Machine (JVM) must be present. A DBMS vendor may provide a specialized JVM with its product, or it may allow the use of any JVM that supports the operating system that is being used.

The classes that we have imported may contain many methods. In SQLJ Part 1, we will make direct use of only methods that are both public and static. Static methods execute independently of any specific object instance. Like all methods, these methods may return values of some data type or they may be void.

So, we have Java public static methods, a JVM in which they can execute, and the desire to invoke them from within SQL. A decision that could have been made would have been to create new SQL statements to invoke these methods. Instead, the decision was made to provide a mapping from SQL routines to Java methods. In that way an application can invoke these routines in exactly the same way that it would invoke SQL's other types of routines. A *Java routine* is a routine that has a mapping to a Java method that provides its implementation.

Mappable Data Types

For an SQL routine invocation to result in a Java method invocation, the arguments to the SQL routine must become arguments to the Java method. It makes sense that SQLJ Part 1 uses the mapping that JDBC defines between SQL data types and Java data types.

You may remember that Java provides some primitive data types that have class wrappers associated with them. An SQL data type and a Java data type are *mappable* if they are *simply mappable* (to Java primitive data types), *object mappable*, or *output mappable* (which we'll discuss when we get to Java procedures).

SQL data type	Simply Mappable	Object Mappable
CHAR	–	String
VARCHAR	–	String
LONGVARCHAR	–	String
NUMERIC	–	java.math.BigDecimal
DECIMAL	–	java.math.BigDecimal
BIT	boolean	Boolean
TINYINT	byte	Integer
SMALLINT	short	Integer
INTEGER	int	Integer
BIGINT	long	Long
REAL	float	Float
FLOAT	double	Double
DOUBLE	double	Double
BINARY	–	byte[]
VARBINARY	–	byte[]
LONGVARBINARY	–	byte[]
DATE	–	java.sql.Date
TIME	–	java.sql.Time
TIMESTAMP	–	java.sql.Timestamp

("–" indicates the same value that is found in the Object Mappable column)

Java Functions

SQLJ Part 1 defines a new variant of `CREATE FUNCTION` that provides a mapping between an SQL function signature and a Java method.

A SQL reference to a specific Java method is made up of three parts:

jar name : *method name* (*signature*)

The Jar name is the SQL name that has been given to the Jar file, the method name contains the name of the method, along with the class and possibly the package that contains it. The third part of the reference, the signature, is optional. If it is not provided, then a signature containing the default Java data type for each SQL data type in the SQL signature is used.

We define a Java function in the following way:

```
CREATE FUNCTION j_pub_year (INTEGER volume)
      RETURNS CHAR (4)
EXTERNAL NAME 'sigmod_jar:pubYear'
LANGUAGE JAVA
PARAMETER STYLE JAVA ;
```

In order to create a Java function, the SQL parameters must be mappable to the corresponding Java method parameters, the result types of the two must be mappable, and the method must be both public and static (as we mentioned earlier).

Once the Java function has been created, it is invoked just like any other SQL function.

```
SELECT   *
FROM     sigmod_articles sa
WHERE    j_pub_year (sa.volume) = '1995';
```

Java Procedures

The parameter mapping that we saw in Java functions must be extended for the `INOUT` and `OUT` parameters that an SQL procedure may contain. This is a bit of a problem, as Java does not directly provide any way for a method to return more than one value as the result of its execution.

Java does, however, allow the state of objects that are passed to a method to be changed. SQLJ Part 1 adopts the convention that a Java method must use a one element array to pass the values of `OUT` or `INOUT` parameters. The single element of the array may be read by the method for its input value and set by the method to indicate its output value. This leads us to the third form of mappable that we mentioned earlier. An SQL data type is output mappable to an array of the Java types that it is simply mappable to or object mappable to. This means that an SQL `INTEGER` type is output mappable to either `int[]` or `Integer[]`.

Let us say that the following method, which provides a Java implementation of the `pub_info` SQL procedure we saw earlier, was included in our `Sigmod` class:

```
public static void pubInfo (int volume,
                            int no,
                            String[] yyyy,
                            String[] mmm) {
  yyyy[0] = pubYear(volume);
  if (no == 1) mmm[0] = "MAR";
  else if (no == 2) mmm[0] = "JUN";
  else if (no == 3) mmm[0] = "SEP";
  else if (no == 4) mmm[0] = "DEC";
  else mmm[0] = "???";
}
```

This method can be used to define a Java procedure in the following way:

```
CREATE PROCEDURE j_pub_info
                     ( IN INTEGER volume,
                       IN INTEGER no,
                       OUT CHAR(4) yyyy,
                       OUT CHAR(3) mmm)
EXTERNAL NAME
  'sigmod_jar:pubInfo (int, int, String[],
                        String[])'
LANGUAGE JAVA
PARAMETER STYLE JAVA ;
```

In order to create a Java procedure, the SQL `IN` parameters must be mappable to the corresponding method parameters, the SQL `INOUT` and `OUT` parameters must be output mappable to the corresponding method parameters, and the method must be declared as void.

The Java arrays are not seen by the SQL `CALL` statement. SQL automatically creates and populates the Java array when the invocation takes place, and it retrieves the necessary elements when the invocation has ended.

Once again we see that the invocation of a Java procedure is indistinguishable from that of an SQL procedure.

```
CALL j_pub_info (22, 3, p_year, p_month);
```

Exception Handling

Within the Java method that underlies a Java routine, exceptions may be thrown and caught in the usual manner. If an exception is not caught by the Java method, then an SQL exception is raised by the SQL statement that invoked the Java routine.

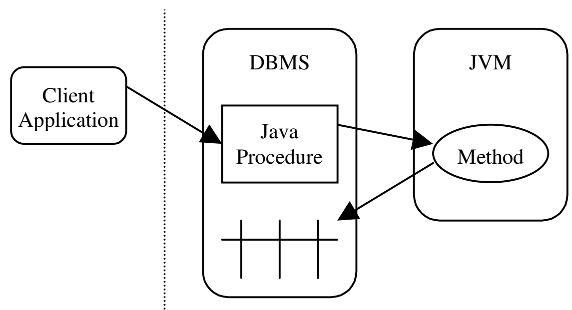
Database Access from within a Java Method

Let's review the environment is which all of this SQL and Java invocation is taking place. An SQL statement is executing on behalf of some user, possibly on a client machine (via an ODBC call or a JDBC method invocation). The SQL statement causes the invocation of a Java method in a JVM.

Some methods may wish to make calls to a database as part of their execution. To do this, they would contain JDBC method invocations.

These JDBC statements could attempt to connect to some specific database by specifying its URL in the `getConnection` method. A DBMS may or may not allow such a connection to be established. A DBMS must, however, allow a Java routine to connect to the database that invoked it with the special URL `jdbc:default:connection`. This connection allows the Java method to share the session and transaction context of the statement that invoked it.

Simply put, this means that the execution of an SQL statement may cause the execution of Java methods, and while the statement and these methods are executing the methods can be making calls back to the database.



Java Procedures Returning Result Sets

Earlier, we discussed how the `article_count` SQL procedure could return one or more result sets along with its `OUT` or `INOUT` parameters. Java procedures may also return result sets by creating JDBC `ResultSet` objects and returning them to SQL. This can be seen in the following example:

```
public static void ArticleCount
                      (int volume,
                       ResultSet[] rs)
     throws SQLException {
  String query
          =   "SELECT   no, COUNT(*) "
            + "FROM     sigmod_articles sa "
            + "WHERE    sa.volume = ? "
            + "GROUP BY no";
  Connection conn
     = DriverManager.getConnection
          ("jdbc:default:connection");
  PreparedStatement pstmt
           = conn.prepareStatement(query);
  pstmt.setInt(1, volume);
  rs[0] = pstmt.executeQuery();
}
```

```
CREATE PROCEDURE j_article_count
                     (IN INTEGER volume)
     READS SQL DATA
     DYNAMIC RESULT SETS 1
EXTERNAL NAME
  'sigmod_jar:ArticleCount
                 (int, ResultSet[])'
LANGUAGE JAVA
PARAMETER STYLE JAVA ;
```

This example could be rewritten to use SQLJ Part 0 statements instead of JDBC methods. In this case, SQLJ Part 1 allows an `sqlj.runtime.ResultSetIterator` to be returned instead of a `ResultSet`.

SQL-Java Paths

The invocation of a Java routine causes the associated Java method to be invoked. This method has been unambiguously identified in the `CREATE FUNCTION` or `CREATE PROCEDURE` statement that created it, with a Jar name, method name, and signature. It is very likely that these methods will themselves contain Java method invocations. In most JVM's, a classpath environment variable exists to determine the order in which directories, Jar files, and Zip files are searched for a method.

In SQLJ Part 1, an *SQL-Java Path* exists for each Jar file to determine the order in which Jar Files are searched when a method invocation takes place. The `sqlj.alter_java_path` procedure exists to allow the SQL-Java path to be modified.

```
CALL sqlj.alter_java_path
  ('sigmod_jar',
   '(Utility/Soundex, util_jar)
    (Sigplan, sigplan_jar)'
  );
```

The second argument of this procedure contains pairs of class name patterns and Jar names. When a method is invoked, the first pair that contains a pattern that matches the class name of the method is used to determine which Jar file to use.

Once this statement has been executed, the invocation of the `Utility.Soundex.isLike` method by a method in the `sigmod_jar` Jar file would locate the method in `util_jar`, even if `sigplan_jar` contains a method by the same name.

Deployment Descriptors

In order for SQL statements to use Java routines, the following steps must be taken:

- Install a Jar file
- Create Java routines that correspond to the public static methods
- Grant access to the Java methods

Deployment descriptors allow the steps that follow the installation of the Jar file to be defined in the Jar file itself and executed automatically. A deployment descriptor contains a number of actions to take when the Jar file is installed, and a number of actions to take when the Jar file is removed. Since the SQL name of the Jar file is not known when the deployment descriptor is written, the placeholder `thisjar` is used to identify Java methods. We might create the file `sigmod_deploy.txt`, with the following text, and place it in our Jar file.

```
SQLActions[] = {
   "BEGIN INSTALL
       CREATE FUNCTION j_pub_year
                         (INTEGER volume)
          RETURNS CHAR (4)
       EXTERNAL NAME 'thisjar:pubYear'
       LANGUAGE JAVA
       PARAMETER STYLE JAVA ;

       GRANT EXECUTE ON j_pub_year
                     TO PUBLIC;
    END INSTALL",

   "BEGIN REMOVE
       REVOKE EXECUTE ON j_pub_year
                      FROM PUBLIC RESTRICT ;
       DROP FUNCTION j_pub_year RESTRICT ;
    END REMOVE"
}
```

This file would then be identified by following manifest entry in our Jar file:

```
Name: sigmod_deploy.txt
SQLJDeploymentDescriptor: TRUE
```

The following `sqlj.install_jar` procedure, because it uses 1 as its last argument, will execute the statements in the deployment descriptor.

```
CALL sqlj.install_jar
            ('file:///d:/Sigmod.jar',
             'sigmod_jar',
             1);
```

Summary

The design of SQL/PSM took a database-centric view of the world. It was recognized that behavior was required in the database, so a procedural language was created for that purpose (Oracle has PL/SQL, Sybase has TransactSQL, and Informix has Stored Procedure Language, or SPL). These languages were designed to work seamlessly with SQL. They support SQL's data types, null values, and SQL's exception handling. Because memory management and pointers were not provided, these languages could not compromise the safety and robustness of the DBMS.

It was also recognized that users had bodies of existing code that they wished to be able to use, and so external routines were provided. This mechanism satisfies a user's preference for a specific language, and can be advantageous when some routines are just too hard to write in SQL/PSM.

SQLJ Part 1 allows a database designer to reuse Java classes that may have been initially written for a client application or an application server. For Java routines that are written specifically for a database, the database designer can leverage his or her training in Java, and take advantage of the development environments and tools that exist for Java. Unlike external routines in other 3GL's, safety and robustness are not compromised by the use of Java routines.

References

[1] *SQLJ Part 0, now known as SQL/OLB (Object-Language Bindings)*, Andrew Eisenberg and Jim Melton, ACM SIGMOD Record, Dec. 1998.

[2] ANSI X3.135.10:1998, *Database Language SQL — Part 10: Object Language Bindings (SQL/OLB)*, 1998.

[3] ANSI NCITS 331.1-1999, *SQLJ — Part 1: SQL Routines using the Java™ Programming Language*, 1999.

[4] ISO/IEC 9075:1996, *Information technology—Database languages—SQL—Part 4: Persistent Stored Modules (SQL/PSM)*, 1996.

[5] *New Standard for Stored Procedures in SQL*, Andrew Eisenberg, SIGMOD Record, Dec.1996

Acquiring the SQLJ Specifications

The SQLJ Part 1 specification will be available in the United States from:

American National Standards Institute
Attn: Customer Service
11 West 42nd Street
New York, NY 10036
USA

Phone: +1.212.642.4980

It can be ordered online at:

http://www.cssinfo.com/ncits.html

Web References

American National Standards Institute (ANSI)
http://web.ansi.org

National Committee for Information Technology Standards (NCITS)
http://www.ncits.org

NCITS H2 – Database Committee
http://www.ncits.org/tc_home/h2.htm

SQLJ
http://www.sqlj.org/