SQLJ Part 0, now known as SQL/OLB (Object-Language Bindings)

Andrew Eisenberg
Sybase, Burlington, MA 01803
andrew.eisenberg@sybase.com

Jim Melton
Sybase, Sandy, UT 84093
jim.melton@sybase.com

Introductions

For about a year and a half, an informal and open group of companies has been meeting to consider how the Java™ programming language and relational databases might be used together. Initially called JSQL and later SQLJ, the companies that have participated in this group are Compaq (Tandem), IBM, Informix, Micro Focus, Microsoft, Oracle, Sun, and Sybase.

The intent of this group when it was formed was to suggest and review one another's ideas, meeting fairly often, see where there was common understanding and agreement on syntax and semantics, and to eventually provide a basis for one or several formal standards.

The work began with a proposal on how SQL statements might be embedded in Java, put forward by Oracle. Later Sybase put forward proposals on how to use Java in the database to provide the implementation of stored routines and user-defined data types (UDTs). Once an initial draft of a specification was put forward, the entire group participated in reviewing it, spotting problems, and suggesting enhancements. These 3 parts are, then, roughly described as:

Part 0	Embedded SQL in Java
Part 1	Java Stored Routines
Part 2	Java Data Types

These three parts have all progressed rapidly. Since work on Part 0 was started before the other parts, it was the first to be submitted to a formal standards body. SQLJ Part 0 has been processed as Database Language SQL — Part 10, Object Language Bindings (SQL/OLB), by NCITS H2, the Database Language technical committee. The name for this part of the SQL standard implies a broad scope, with SQLJ Part 0 being the first such binding to be defined; although there are presently no proposals to extend this to other languages, several participants have expressed interest in supporting other object-oriented languages, such as C++ or Smalltalk.

As this article is being written, the editor has just submitted his resolution of comments for the U.S. public review that just took place. It is expected that SQL/OLB will be formally approved by the end of 1998. When it is approved, it will be available for purchase from ANSI as ANSI X3.135.10:1998.

Javasoft's JDBC™

JDBC, initially provided in JDK 1.1, defines a Java API for accessing relational DBMSs. It is mentioned here because SQLJ Part 0 layers upon it.

The JDBC API is a fairly rich one. It provides classes and methods to:

- connect to the database
- get capability, syntax, and limit metadata from the database
- execute a query or DDL statement
- prepare a DML or call statement, with parameters to be supplied at the time the statement executes
- retrieve both data and metadata for result sets produced by statement execution

JDBC provides for the dynamic execution of SQL statements. Any syntax or semantic errors in the SQL statements will raise exceptions at the time the application runs.

A JDBC driver must support Entry SQL-92 statements, with some extensions defined in the JDBC specification. It may also support statements from other levels of SQL and statements that are vendor-extensions to the SQL standard.

A very straightforward program to retrieve some data from a table might look like this:

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

```
try {
   String url
      = "jdbc:sybase:Tds:localhost:2638";
   Connection con =
       DriverManager.getConnection
                       (url, "DBA", "sql");

   String stmt_source =
           "SELECT distinct city, state "
         + "FROM   employee";

   Statement stmt = con.createStatement();
   ResultSet rs
        = stmt.executeQuery(stmt_source);

   while (rs.next()) {
      System.out.println
               (rs.getString("city")
                + " " + rs.getString(2)
               );
   }
   con.close ();
}
catch (SQLException sqe) {
   System.out.println (sqe.getMessage());
}
```

SQLJ Part 0 Features

SQLJ Part 0 allows SQL static statements to be embedded in a Java program, in somewhat the same way that SQL-92 allows SQL statements to be embedded in C, COBOL, and several other languages. Dynamic SQL statements are handled just fine by JDBC, and so were not included in this effort.

A simple SQLJ Program

The following code fragment shows how SQLJ Part 0 can be used to access a database:

```
try {
  #sql { DELETE
         FROM     employee
         WHERE    emp_id = 17
       };
}
catch (SQLException sqe) {
   System.out.println
                 (sqe.getMessage());
}
```

This example does not show some of the setup that is necessary for this example to be used. A JDBC driver must be registered with the JDBC Driver Manager, and a connection must be made to a database.

"`#sql { ... };`" identifies an SQL executable statement. The curly braces ({ }) have been used to delimit the SQL statement and separate it from the rest of the Java program. An SQLJ translator will look for these embedded statements, and replace them with Java statements that cause the SQL statements to be executed. The resulting Java source program will be compiled normally.

At the time that the SQLJ translator runs, it can be told to connect to an *exemplar database* and use the metadata it finds there to validate the SQL statements. If the `employee` table did not exist, or the `emp_id` column did not exist, then the SQLJ translator would notify the user of the error.

It is also possible that the vendor of the SQLJ translator will provide off-line checking, which could do the part of the checking of the statement that does not require metadata. If the "=" in the above statement were a "+" instead, then off-line checking would be able to catch this mistake.

At execution time, the application can connect to the same database against which it was checked, or to another database with the same schema.

The example above also shows that the JDBC model for dealing with SQL exception conditions has been used. In other host language bindings, the SQLSTATE variable is used to inform the application of a SQL exception condition. In SQLJ Part 0, such an exception condition will cause the SQLJ statements to throw a Java exception, `java.sql.SQLException`.

In the sections below, we will make this example more complicated to show additional SQLJ Part 0 features.

Connection Contexts

A *connection context* object is used to associate the execution of an SQL statement with a particular connection to a database. In the example above, an implicit connection context object was used. In the following example an explicit connection context object will be used.

```
#sql context EmpContext;

String url
    = "jdbc:sybase:Tds:localhost:2638";
EmpContext empCtxt =
    new EmpContext(url, "dba",
                  "sql", false);

#sql  [empCtxt] { DELETE
                  FROM     employee
                  WHERE    emp_id = 17
                };
```

The example begins with "`#sql context ...`" to declare of the connection context class (a subclass of ConnectionContext), called EmpContext. Later, an empCtxt object of this class is created. It appears in square brackets ([ ]) to indicate its use in the execution of the DELETE statement.

A connection context class is used by an SQLJ translator to determine which database schema is to be used to check the validity of all of the statements that specify connection context objects of that class.

This type of explicit context is more portable than the use of an implicit context. The connection context class provides methods that can examine and change some of the properties of the connection.

An application can operate on multiple connections to the same database, or multiple connections to different databases, using explicit connection context objects. Connection contexts may be safely shared among threads in a multithreaded application.

In some environments, such as within a DBMS, an SQLJ application may be invoked with a connection context already provided for its use. The `ConnectionContext.getDefaultContext()` method can be used to determine if this is the case.

Execution Contexts

An *execution context* object allows some aspects of a statement's execution to be controlled, and it allows the retrieval of information about the execution after it has completed.

```
sqlj.runtime.ExecutionContext execCtxt
    = new sqlj.runtime.ExecutionContext();

#sql  [empCtxt, execCtxt]
             { DELETE
               FROM     employee
               WHERE    emp_id = 17
             };

System.out.println
       ( "Deleted "
        + execCtxt.getUpdateCount()
        + " rows."
       );
```

In this example, an explicit connection context and execution context are associated with the DELETE statement. The ExecutionContext object empCtxt is used to access the number of rows that were effected by the delete. Any warnings that were generated by the statement can be retrieved from the execution context using the `getWarnings()` method.

Like a connection context, the use of an execution context can be implicit or explicit. A connection context has a default execution context associated with it. This default execution context is used if an explicit execution context has not been specified. The default execution context can be retrieved using the `ConnectionContext.getExecutionContext()` method.

Unlike a connection context, an execution context should not be shared among threads in a multithreaded application.

Host Variables and Expressions

Ordinary static SQL allows for the use of host variables in expressions, in addition to literals, column references, SQL variables, and SQL parameters.

SQLJ Part 0 allows the use of Java host variables and host expressions, as seen in the following example:

```
int id;

#sql { SELECT   emp_id
       INTO     :id
       FROM     employee
       WHERE    emp_fname
                     LIKE :(argv[0] + '%')
     };

System.out.println ("Employee id " + id);
```

The Java host variable, "`id`" in this example, is used within an SQL statement to indicate that a value must be placed into a Java variable. The pattern in the LIKE clause is a host expression that concatenates an element of a String array with a String literal. SQLJ uses the same mapping between Java data types and SQL data types that is defined by JDBC.

The syntax for a host variable is as follows:

```
<host expression> ::=
   : [ <parameter mode> ] <expression>

<parameter mode> ::= IN | OUT | INOUT

<expression> ::=
     <variable>
   | ( <complex expression> )
```

Java variables and host expressions have a parameter mode that—if not explicitly specified—is implicitly determined by its use. The host variable "`id`" has mode OUT and host expression in the LIKE clause has mode IN.

Side effects are a possibility when Java expressions are evaluated—for example, if an expression such as "`count++`" appears inside a <host expression>, the value of "`count`" will be incremented by one every time the host expression is encountered. SQLJ evaluates all of the Java expressions that appear in a SQL statement before the statement is executed. The expressions are evaluated in left-to-right order.

Calling Stored Routines

A stored procedure can be invoked—using the SQL CALL statement—as follows:

```
int count = 0;

#sql { CALL emp_count (:in (argv[0]),
                       :in (argv[1]),
                       :out count) };

System.out.println
        ("The result is " + count);
```

A stored function can be invoked as follows:

```
int count = 0;

#sql count = { VALUES (emp_count2
                          (:in (argv[0]),
                           :in (argv[1])
                          )
                      )
             };

System.out.println
        ("The result is " + count);
```

Result Set Iterators

By far the most often used statement in applications is the SELECT statement. A *result set iterator* is roughly comparable to an SQL cursor; it is used to access the rows of the result of a query. Because a result set iterator is a Java object in SQLJ, it can be passed as an argument in the invocation of a method.

SQLJ Part 0 provides two types of result set iterators. These two types cannot be intermixed for a single result set; one or the other must be chosen.

Binding to Columns by Name

The first of these types, the *named iterator*, is seen in the following example:

```
#sql iterator Employee
                (int              emp_id,
                 String           emp_lname,
                 java.sql.Date    start_date
                );

Employee emp;

#sql emp = { SELECT   emp_lname, emp_id,
                         start_date
             FROM     employee
             WHERE    emp_fname LIKE 'C%'
           };

while (emp.next()) {
   System.out.println
          (emp.start_date() + ", "
           + emp.emp_id() + ", "
           + emp.emp_lname().trim()
          );
}

emp.close();
```

The SQLJ Part 0 iterator statement "`#sql iterator ...`" defines an Employee class with accessor methods for each of the columns of the query result. Methods like `next()` and `close()` are also generated. The emp object is created and bound to the result of the query. The `while` loop then iterates over the rows that result from the query, printing a line for each one. Finally, the result of the query is closed.

It is the use of column name/data type pairs in the iterator declaration that determines that this is a named iterator. The column names are matched to the iterator names in a case-insensitive way. This means that SQLJ Part 0 requires the select list columns to be unique when they are compared in this way (without regard to case).

Binding to Columns by Position

The second type of result set iterator that is used to access the result of query is the *positioned iterator*, as shown in the following example:

```
#sql iterator Employee
                 (int, String, String);

int emp_id = 0;
String emp_lname = null;
String emp_fname = null;

Employee emp;

#sql emp = { SELECT   emp_id, emp_lname,
                         emp_fname
             FROM     employee
             WHERE    emp_fname LIKE 'C%'
           };

while (true) {

   #sql {FETCH :emp INTO :emp_id,
                         :emp_lname,
                         :emp_fname};

   if (emp.endFetch()) break;

   System.out.println
               (emp_fname.trim() + " "
                + emp_lname.trim() + ", "
                + emp_id
               );

}

emp.close();
```

The iterator statement in this example has only data types, rather than column name/data type pairs, indicating that positioned iterator has been chosen. At the time the fetch is done, the result columns are stored in the Java variables, in the order that they were both specified. The Employee class has methods like `endFetch()` and `close()` generated for it. The application traverses the rows of the query result with the SQL `FETCH` statement.

The choice between named iterator and the positioned iterator is entirely a stylistic one. To use the named iterator, the columns that are selected must all have unique names, or they must be given unique names using SQL's ability to rename columns with the `AS` clause.

Positioned Update and Delete Statements

The use of the positioned UPDATE statement can be seen in the following example:

```
#sql iterator Employee
implements sqlj.runtime.ForUpdate
(String emp_lname, String emp_fname);

Employee emp;

#sql  emp = { SELECT   emp_lname, emp_fname
              FROM     employee
              WHERE    dept_id = 200
            };

while (emp.next()) {

   System.out.println(emp.emp_fname()
                     + " "
                     + emp.emp_lname()
                    );

   #sql { UPDATE   employee
          SET      dept_id = 100
          WHERE    CURRENT OF :emp
        };

}
```

The use of "`implements sqlj.runtime.ForUpdate`" is required to indicate to SQLJ Part 0 that this an updatable result set iterator—without this clause, the iterator would be read-only. The SQL positioned `UPDATE` statement has been changed only slightly in SQLJ Part 0. Instead of specifying a cursor name in the "`WHERE CURRENT OF`" clause, the Java iterator variable is used.

Multiple Result Sets

Several DBMSs allow a stored procedure to return result sets during the execution of the procedure. These are sometimes called *side-channel* result sets.

These result sets are dynamic in nature. Two invocations of the same procedure could result in different numbers of result sets being returned, or the same number of result sets but with different structures.

SQLJ allows an application to deal with these result sets by escaping to JDBC. This can be seen in the following example:

```
ExecutionContext ectxt
   = new ExecutionContext();

#sql [ectxt]
        { CALL emp_count3 (:in (argv[0]),
                           :in (argv[1])
                          ) };

ResultSet rs;
while ((rs = ectxt.getNextResultSet())
       != null) {
   while (rs.next()) {
      System.out.println (rs.getString(1));
   }
   System.out.println();
}
```

`ResultSet` is the JDBC interface that allows an application to process result sets.

SQLJ/JDBC Interoperability

As we have stated more than once, JDBC was designed to process dynamic SQL statements and SQLJ Part 0 was designed to process static SQL statements. Clearly there are some applications that will need to do both types of processing. SQLJ Part 0 provides interoperability between itself and JDBC through the use of connection contexts and result set iterators.

A connection context may be created using a URL, as we showed earlier, or it may be created by specifying a JDBC connection. Going the other way, the `getConnection()` method may be applied to a connection context in order to obtain the JDBC connection it is using.

An SQLJ iterator may be created from a JDBC result set as follows:

```
ResultSet rs = ... ;
#sql iterator iter (...);

#sql iter = { CAST :rs }
```

Going the other way, the `getResultSet()` method may be applied to an SQLJ iterator, returning the JDBC result set associated with the iterator.

Binary Portability

Java programs are independent of the hardware platform on which they run. As SQLJ programs are turned into pure Java, they can run anywhere that a Java Virtual Machine exists.

Similarly, JDBC allows applications to be written that are independent of the DBMS that will be used at execution time. Because SQLJ generates JDBC calls, a single SQLJ application could be run against multiple DBMSs.

SQLJ provides an additional level of independence from the DBMS. The SQL statements that have been processed by SQLJ are accessible to a vendor-specific *customizer*. This customizer might, for example, generate code to create and to execute stored procedures contain the application's original SQL statements. This generated code will become part of the SQLJ application. At runtime if a customization for the connected DBMS exists, then the code generated by that customizer will be used. If no such customization is found, then the original JDBC code will be used.

Levels of Conformance

A SQLJ implementation is s*pecification conformant* if it supports the syntax and semantics specified in SQL/OLB, supports JDBC 1.2 or higher, and supports the SQL language specified in JDBC 1.2.

A SQLJ implementation is *Core SQLJ Conformant* if it meets these requirements, with the exception of a small number of features, such as:

- calls to stored procedures and functions
- some of the ExecutionContext attributes
- `getResultSet()` and `getJDBCResultSet()` methods
- SQL `begin/end` statements

Advantages of SQLJ Part 0

We began this discussion with an example of JDBC and indicated that JDBC was designed to process dynamic SQL statements. What then are the advantages to using SQLJ Part 0 over JDBC for static statements?

- SQLJ Part 0 statements are amenable to translator-time validity checking. An off-line checker could check some of the SQL syntax. An on-line checker could check all of the SQL syntax and semantic rules.
- SQLJ Part 0 statements and programs are generally shorter and more easily read than the JDBC equivalents.
- SQLJ Part 0 allows a DBMS vendor to offer tools to customize the SQLJ application, optimizing it in ways that would be impractical for a JDBC application.

SQLJ Part 0 Reference Implementation

Oracle, when it introduced SQLJ Part 0 to the SQLJ group, also made available a reference implementation of this technology. This can be found at Oracle's web site.

This reference implementation is itself written in Java, so that it can be run on any platform that supports a JVM. It is vendor neutral. The online syntax checker uses a JDBC connection to validate SQL statements. An offline checker can be invoked if a JDBC connection is not available. The offline checker is implemented by a Java class, which can be written by the vendor of a DBMS.

SQLJ Part 1 and Part 2

SQLJ Parts 1 and 2 will soon be sent to NCITS for adoption as formal standards (though not as parts of the SQL standard). An SD-3 (project proposal) has been submitted to NCITS requesting Fast-Track processing for these specifications. It is possible that these specifications will be formally adopted during the 2nd half of 1999. We will very likely devote a column to them at that time.

Recognition of Individual Contributors

Many people have contributed to the SQLJ Part 0 specification. At the risk of missing someone, we will try to list these contributors: Julie Basu, Brian Becker, David Birdsall, José Blakely, Charles Campbell, Gray Clossman, Curt Cotner, Paul Cotton, Dan Coyle, Stefan Dessloch, Pierre Dufour, Cathy Dwyer, Andrew Eisenberg, John Ellis, Chris Farrar, Mark Hapner, Johannes Klein, Jim Melton, Chong-Mak Park, Frank Pellow, Richard Pledereder, Ekkehard Rohwedder, David Rosenberg, Jerry Schwartz, Phil Shaw, Yah-Heng Sheng, Ju-Lung Tseng, Michael Ubell, and Seth White.

References

[1] *dpANS X3.135.10:1998, draft proposed American National Standard, Information Technology — Database Language — SQL — Part 10: Object Language Bindings (SQL/OLB),* June 1998.

[2] *SQL Routines using the Java™ Programming Language, Working Draft,* Oct. 14, 1998.

[3] *SQL Types using the Java™ Programming Language, Working Draft*, Oct. 14, 1998.

The SQL/OLB specification will be available from:

American National Standards Institute
Attn: Customer Service
11 West 42nd Street
New York, NY 10036
USA
212-642-4980

Web References

American National Standards Institute
http://web.ansi.org

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

SQLJ Home Page
http://www.sqlj.org

Oracle, JDBC Drivers Page
http://www.oracle.com/st/products/jdbc/sqlj/