Using SAS ODBC with Java

Lei Zhang, TianShu Li, Merch, Rahway, USA

ABSTRACT

SAS ODBC provides a low cost and convenient way for external applications to directly access SAS metadata and data. This paper discusses how to use Java JDBC to access SAS libraries through SAS ODBC driver. It first describes basic JDBC programming concepts, and then presents two examples to illustrate how to connect with SAS ODBC driver, view and update SAS datasets by submitting SAS-supported SQL queries and commands through the connection. This paper also explores the strengths and limitations of using SAS ODBC with Java.

INTRODUCTION

Java is one of key technologies to Internet/Intranet applications because of its network-oriented nature. SAS Integration Technology offers Java tools as the components of SAS/IntrNet software for Java programs to access the SAS system. The SAS/IntrNet has to be purchased and installed separately; however, by integrating existing SAS ODBC Driver with JDBC, Java programs can read and write SAS data libraries with little or no effort. Therefore, programmers can benefit from both SAS and Java worlds without any extra large cost. This paper describes the use of SAS ODBC and JDBC to access the SAS libraries and discuss its features and limitation. It is presumed that readers have general knowledge of SAS, SQL and Java.

SAS ODBC AND JDBC

ODBC (Open Database Connectivity) is an open standard that provides a common API (Application Programming Interface) for accessing databases. The SAS Institute provides ODBC support with Base SAS for Windows since the Release 6.10[2]. The SAS ODBC driver is an implementation of open ODBC standard that enables users to manipulate and update SAS data sets from ODBC-compliant applications. It supports same SQL grammar as used in Proc SQL[1][2].

JDBC, which stands for Java Database Connectivity, is the Java API for communicating with databases. It specifies the interface necessary to connect to a database, execute SQL commands and queries, and interpret the results. This interface is both database-independent and platform-independent. JDBC bases its framework on ODBC. Both of them provide vendor-transparent access to databases through a standard interface; However, JDBC has higher level abstraction than ODBC, and has the added advantage of being platform independent.

JDBC only defines a database-independent interface and a collection of helper classes for handling results and errors. It is part of the built-in java.sql package. The actual database access is provided by a JDBC driver, a database-specific class that implements the JDBC interface defined by the API. The JDBC drivers are provided largely by database vendors, or a third-party provider, usually at an additional cost. In order to leverages the large number of ODBC-accessible databases available, Sun MicroSystem has provided a free, platform-independent JDBC-ODBC bridge driver in JDK 1.1 or higher that can communicate with any ODBC-compliant relational databases[3]. Even though SAS is not a typical relational database, it is ODBC-compliant. Its ODBC driver comes with your SAS CD-ROM, or can be downloaded from http://www.sas.com. Therefore your Java programs can access SAS libraries by combining SAS ODBC with JDBC-ODBC bridge driver without installing any proprietary JDBC driver or at any cost.

THE BASICS OF JDBC PROGRAMMING

Before using any JDBC programs to access SAS data, what you need to do is to set up a proper JDBC programming environment. First of all, you have to install SAS 6.10 or higher and the corresponding SAS ODBC driver on your local and/or remote machine. Second, you need follow the instructions in "SAS ODBC Driver User's Guide and Programmer's Reference" to establish a SAS ODBC data source name (DSN). You can associate multiple SAS data libraries with a DSN. Finally, you also need download JDK 1.1 or higher from http://java.sun.com and install Java and JDBC on your machine. The JDK you downloaded will include both JDBC and JDBC-ODBC Bridge driver.

In the next sections, we will describe how to open a connection with SAS database, and then demonstrate how to use JDBC to pass SQL statements to the SAS System and process the results that are returned.

ESTABLISHING A CONNECTION WITH SAS DATABASE

The first thing you need to do is to establish a connection with the SAS database. This involves two steps: (1) loading the JDBC-ODBC Bridge driver and (2) making the connection.

1. Loading the JDBC-ODBC Bridge driver

You have to specify the Sun's JDBC-ODBC driver in your JDBC program and load it with the following code:

```
Class.forName("sun.jdbc.odbc.JdbcOdbcDriver");
```

The Class.forName() method makes the JVM (Java Virtual Machine) load the driver class into memory. When the driver is loaded into the JVM's memory, it registers itself with JDBC's java.sql.DriverManager class. You then use static methods of DriverManager to obtain a reference to a driver-specific Connection object, which in turn provides access to the database, as explained next.

2. Making the Connection with SAS database

Once you have successfully loaded the JDBC-ODBC driver, you're ready to make a SAS database connection, which will provide you with an instance of the `Connection` class. JDBC follows the Java paradigm of using URL to connect to a data resource. The format of the URL for JDBC-ODBC bridge driver like this:

```
Jdbc:odbc:data-source-name
```

The URL for the JDBC connection is passed to one of the Driver Manager's getConnection() methods. Those static methods accept several different sets of arguments, depending on whether you need a user and password to connect to the SAS database. Since there is a lot happening while loading driver and creating connection, the Java compiler will require you to take possible exceptions into account by using the proper `try/catch` blocks. Thus, the code snippet to establish the connection to a SAS ODBC data source named *mySASDSN* would look like the following:

```
Connection  conn;
Try {
     Class.forName(
"sun.jdbc.odbc.JdbcOdbcDriver");
String url = "jdbc:odbc:mySASDSN";
```

```
conn = DriverManager.getConnection(url,
"user", "password");
} catch (ClassNotFoundException e) {
    System.err.println("Could not load JDBC-
ODBC bridge driver!");
} catch (SQLException e) {
    System.err.println("Could not connect to
SAS database!");
} finally {
    try { if (conn != null) conn.close();}
    catch (SQLException ignore) {}
}
```

Note that we here use a `finally` block, which in turn requires its own `try/catch` block, to make sure that the SAS database connection is closed at the end of the program. Once you have a valid Connection object, you can issue SQL queries and commands to the SAS database and manipulate the results.

RETRIEVING DATA FROM A SAS TABLE

Before you can issue any SQL queries and commands to the SAS database through JDBC, you need to create a `Statement` object. While a `Statement` object can be reused across multiple SQL requests, the object itself is created by the Connection object and is therefore dependent on the JDBC-ODBC driver you're using. Below is a single line code to create `Statement` object.

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

- Issuing a SQL Query to SAS Database through JDBC

Once you have a `Statement` object, you can simply pass a SQL query to the object's executeQuery method. The result is returned as an instance of JDBC's ResultSet object. Here is what a SQL query of the `Dictionary.tables` would look like:

```
String selectSql = "select distinct libname
from dictionary.tables";
ResultSet results =
stmt.executeQuery(selectSql);
```

Please note that you have to use *library-name.table-name* format to reference a table in a SAS library instead of a single table name. This is because a SAS ODBC data source normally has association with multiple SAS libraries. Besides, in the real world the above code snippet would need to be encased in a `try/catch` block to catch any SQL exceptions that will be thrown. The results of this query are stored in the result variables, which contains a list of different SAS library names you can access with the established connection.

The ResultSet object exposes the contents of this list one row at a time and provides methods to access data from each column in the current row, as indicated by an internal cursor. When the result set is first created, the row cursor isn't at the first row, but rather just before it. Calling the next() method of ResultSet advances the cursor one row, returning true if the call was successful. When the internal cursor is positioned at a row you would like to check, you can call any of the dozens of data access methods supported by the Results object to retrieve data from the columns and get information about the results. These methods mostly follow the same naming pattern as `getDatatype()`, for example, getString(), getInteger(), or getDate(). Each method accepts an argument that specifies either the name or the index number of the column you want to examine. The row you read from is determined by where the cursor is currently positioned. For example, you can print the contents of the list in the following code by continually calling the next() method until it returns false, indicating that you've passed the end of the result set:

```
While (results.next()) {
    String libname =
results.getString("libname");
    System.out.println("SAS Library Name: " +
libname);
}
```

You could also have referred to the columns in the ResultSet object by index number rather by name. This can be useful in building applications that aren't dependent on any particular database table schema. In fact, another object that you can create is a ResultSetMetaData object, which contains information about the results themselves. With the ResultSetMetaData object, you can determine the amount and type of data held in the rows and columns of the result set, making it possible to handle completely arbitrary queries and database schemas.

- Inserting, Updating, and Deleting SAS Data

Like retrieving data, inserting, updating and deleting data in the SAS user libraries can be accomplished with SQL commands. You can issue these SQL commands through the Statement object as before, but through excuteUpdate() method. The method doesn't return ResultSet objects; instead, it returns an integer value that specifies the number of rows altered by the SQL command.

APPLICATION EXAMPLES

In this section, we will show two examples of Java programs that connect to an SAS ODBS source, view and manipulate the SAS data sets. The first program that will print a list of different SAS libraries is very terse to ensure that when you run it, there is no possible problem with the program itself. The second program is to transfer data from Microsoft Access to SAS with JDBC-ODBC Bridge driver.

EXAMPLE 1: DISPLAYING A LIST OF SAS LIBRARIES

We choose to write a simple JDBC program called TestSASODBC to show a list of SAS libraries that you can access with a connected SAS ODBC data source named *mySASDSN*. In this case, we are going to query SAS system table *dictionay.tables* for all the libraries available and display the result set to the command line. First, the program imports the JDBC library `java.sql.*`. Next, it uses the `Class.forName()` method to load and register sun's JDBC-ODBC bridge driver. Then, it establishes a connection using the `getConnection(String url, String username, String password )` method. Finally, the program creates an SQL statement to retrieve a list of different SAS libraries available from *dictionay.tables* and display it. Here is the code of the example program:

```
import java.sql.*;
public class TestSASODBC {

public static void main(String args[])
      throws ClassNotFoundException,
SQLException {
      Class.forName(
"sun.jdbc.odbc.JdbcOdbcDriver");
          Connection conn =
DriverManager.getConnection(
            "jdbc:odbc:" + args[0], args[1],
args [2]);
          Statement stmt =
conn.createStatement();
          ResultSet rs   = stmt.executeQuery(
          "select distinct libname from
dictionary.tables");
          while(rs.next())

System.out.println(rs.getString(1));
```

```
            rs.close();
            stmt.close();
            conn.close();
        }
    }
```

To test the program, type the code into a file named TestSASODBC. Then, to compile and execute the program, type the command shown in the following example:

```
C:\>javac TestSASODBC.java
C:\>java  TestSASODBC  mySASdsn user password
```

You should get a list of SAS libraries as you run the program, including system or default data libraries such as MAPS, SASHELP, SASUSER, and SASADMIN. Note that the program doesn't include any exception handling code. Instead, it let the JVM handle any exception that occurs by printing a stack trace.

EXAMPLE 2: TRANSFER DATA FROM MS ACCESS TO SAS

Now that we have executed the TestSASODBC.java program above, and walk though all steps of manipulating SAS libraries using JDBC-ODBC, let's build a little more complicate application. In this case, we will transfer the data from any Microsoft Access table to SAS database. Assuming you have set up ODBC data sources for both Microsoft Access data file and SAS database, the program perform follow steps:

- Open connection to both MS Access ODBC and SAS ODBC data sources
- Read data from a MS Access table with SQL SELECT statement
- Construct a SQL CREATE statement for SAS based on the metadata of the returned ResultSet object from MS Access.
- Create a new SAS table same as MS ACCESS table retrieved by Executing the SQL CREATE Statement on SAS library named *tes*t.
- Loop until the end of the ResultSet:
  - Construct a SQL INSERT statement for SAS based on the meadata and current row of the ResultSet object.
  - Insert a new record into newly-created SAS table by executing the SQL INSERT statement on the SAS library, *test*.
- Clean up by closing all the connections.

The code of the example program is attached in the Appendix. Comments in the code explain the actions of the program.

TIPS, TRICKS AND LIMITATIONS

The few classes and methods we've described here may be sufficient for most database-related application you would write. However, there are a few things that you may have to pay attention to when you use JDBC to access SAS libraries.

First, when your program connects a SAS ODBC data source, your program can not only access the SAS user-defined libraries that are associated with the data source but also can access the SAS system or default libraries such as DICTIONARY, SASUSER, and SASADMIN. Even though you can obtain metadata about the SAS table and system by using typical `connection.getDatabaseMataData()` method, you sometimes need use SQL SELECT statement to access dictionary tables such as dictionary.tables and dictionary.columns to get metadata information specific to SAS tables such as the format, informat of a SAS variable, and the label of the SAS variable.

Second, the SQL queries and commands sent to the SAS database should confirm to the syntax accepted by SAS PROC SQL. You may even send a SQL query in SAS style for the purpose of the performance. For example, you can send following SQL queries to the SAS database through JDBC

```
SELECT * from  Dictionary.tables(drop=format
label);
```

Third, with SAS ODBC driver, JDBC result sets can only use forward-only cursors. That is, there's no previous () method, just like next (). There is no way you can go backwards with ResultSet object.

Fourth, most databases can benefit from Prepared SQL statements and stored procedures, SAS ODBC does not support either. Beside, under SAS, there is no transaction support. That is, you cannot put `commit()` method calls (from Connection class) to delimit your application transactions.

Last, there are some variations in the data type and constraints supported by SAS, so your SQL scripts to create table and so on will have to change accordingly. For example, MS ACCESS uses VARCHAR for string columns, while SAS uses CHAR. MS ACCESS also lets you specify foreign keys, so the database daemon will enforce referential integrity, but SAS won't. These would be subject for another article by itself. So we won't elaborate any more here. Please see reference [1] for more information.

CONCLUSION

SAS ODBC provides an open window to the external applications. By combining SAS ODBC Driver, with JDBC-ODBC bridge driver, it is easy to create powerful JDBC applications to solve a variety of problems with little or no cost. The tool introduced above have much more functionality than it is presented in this paper. Be sure to use it when you want to benefit from both SAS and Java world.

REFERENCES

1. SAS Institute Inc., SAS ODBC Driver Technical Report: User's Guide and Programmer's Reference, Cary, NC: SAS Institute Inc., 1998.

2. Riba, S.David and Riba, Elisabeth. ODBC: Windows to the Outside World, *Proceedings of the Twentieth Annual SAS Users Group International Conference*. SAS Institute, Inc., Cary, NC.

3. Sun Microsystems. *JDBC Data Access API: Articles and Success Stories*
   – http://java.sun.com/products/jdbc/articles.html

TRADEMARK,

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Lei Zhang
Merck Co. & Inc.
RY34-A320
P.O. Box 2000
Rahway, NJ 07065
(732) 594-9856
(732) 594-6075 (Fax)
lei_zhang4@merck.com

Tianshu Li
Merck Co. & Inc.
RY34-A320
P.O. Box 2000
Rahway, NJ 07065
(732) 594-98650378
(732) 594-6075 (Fax)
ltianshu_li@merck.com

APPENDIX

```
import java.sql.*;
public class DataMover{

    private static String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
    private static String sasDSN = "MySASDSN";
    private static String accessDSN = "MyAccessDSN";
    private static String sasLibrary = "test";
    private static String tableName;

    public static void main(String args[])
    throws ClassNotFoundException, SQLException {
        if (args.length != 1) {
            System.out.println("Usage:  java  table_name");
            System.exit(-1);
        }
        tableName = args[0];

        Class.forName(driver); // Load JDBC-ODBC bridge

        // Coonect to MS Access
        Connection conn = DriverManager.getConnection(
        "jdbc:odbc:"+accessDSN, "username", "password");
        // Fetch all the data from the Access table
        Statement stmt = conn.createStatement();
        ResultSet rs = stmt.executeQuery(
        "select * from " + tableName);

        // Connect to SAS
        Connection connSAS =
        DriverManager.getConnection("jdbc:odbc:" + sasDSN);
        Statement stmtSAS = connSAS.createStatement();

        // Get the metadat of the fetched table
        ResultSetMetaData rsmd = rs.getMetaData();
        int numCols = rsmd.getColumnCount();
        // Construct Create SQL statement for the SAS database
        String createSql = getCreateStatement(rsmd);

        //Create the SAS table
        int num = stmtSAS.executeUpdate(createSql);

        while (rs.next()) {
            // Construct the Insert SQL statement for SAS table
            String insertSql = getInsertStatement(rs);
            // Insert a new record into SAS table
            num = stmtSAS.executeUpdate(insertSql);
        }
        rs.close(); stmt.close(); stmtSAS.close();
        conn.close();
    }

    private static String getCreateStatement(ResultSetMetaData rsmd)
    throws SQLException {
        int numCols = rsmd.getColumnCount();
        String columnList = "(";
        for(int i = 1; i <= numCols ; i++) {
            columnList = columnList + rsmd.getColumnName(i) +
            " " + getColumnDef(rsmd, i) + ((i != numCols)?", ":")");
        }
        String sql = "create table " + sasLibrary + "." + tableName +
             columnList;
        return sql;
     }
```

```
    private static String getInsertStatement(ResultSet rs)
    throws SQLException {
            ResultSetMetaData rsmd = rs.getMetaData();
            String valueList = "(";
            int numCols = rsmd.getColumnCount();
            for(int i = 1; i <= numCols; i++) {
                valueList = valueList + getValueDef(rs, i) +
                    ((i != numCols)?", ":")");
            }
            String sql = "insert into " + sasLibrary + "." + tableName +
                    getColumnDef(rsmd) +  " Values " +  valueList;
            return sql;
    }

    private static String getColumnDef(ResultSetMetaData rsmd)
    throws SQLException {
        String  columnDef = "(";
        int n = rsmd.getColumnCount();
        for(int i = 1; i <= n; i++) {
            columnDef = columnDef + rsmd.getColumnName(i) +
                ((i != n)?", ":")");
        }
        return columnDef;
    }

    private static String getColumnDef(ResultSetMetaData rsmd, int i)
    throws SQLException {
        String  columnDef = "";
        String  type = rsmd.getColumnTypeName(i);
        if (type.equalsIgnoreCase("VARCHAR")) {
            columnDef = "CHAR" + "(" + rsmd.getPrecision(i) + ")";
        }else if (type.equalsIgnoreCase("DATETIME")) {
            columnDef = "NUM FORMAT=DATETIME19.";
        }else if (type.equalsIgnoreCase("INTEGER")) {
            columnDef = "NUM ";
        }else if (type.equalsIgnoreCase("DOUBLE")) {
            columnDef = "NUM FORMAT=9.5";
        }else {
            System.out.println("Column Type = " + type);
            columnDef = "NUM FORMAT=9.3";
        }
        return columnDef;
    }

    private static String getValueDef(ResultSet rs, int i)
    throws SQLException {
        String valueDef = "";
        ResultSetMetaData rsmd = rs.getMetaData();
        String type = rsmd.getColumnTypeName(i);
        if (type.equalsIgnoreCase("VARCHAR")) {
            valueDef = rs.getString(i);
            if (valueDef != null)
                valueDef = "'" + valueDef + "'";
            else valueDef = " ";
        }else if (type.equalsIgnoreCase("DATETIME")) {
            valueDef = rs.getString(i);
            if (valueDef != null)
                valueDef = "{ts'" + valueDef + "'}";
            else valueDef = ".";
        }else if (type.equalsIgnoreCase("DOUBLE") ||
        type.equalsIgnoreCase("INTEGER")) {
            valueDef = rs.getString(i);
            if (valueDef == null) valueDef = ".";
        }
        return valueDef;
    }
}
```