Technical Paper

Using SAS® Stored Processes inside Teradata

Table of Contents

Overview

The ability to exercise SAS® functionality against data stored on a Teradata server is an important part of the partnership between SAS and Teradata. There are a few existing methods to achieve this integration. User-defined functions (for example, the Scoring Accelerator) and SAS formats built into the Teradata system allow access to some SAS functionality from within Teradata. The SAS/ACCESS® to Teradata engine enables a user to retrieve data from, and write data to, a Teradata server from within SAS software.

Although both of these approaches are useful, they each have their limitations. User-defined functions and formats enable users familiar with Teradata semantics to access only some of the SAS functionality. The SAS/ACCESS engine enables users to access all SAS functionality against the data, but the user must be familiar with SAS semantics. This prerequisite knowledge can be a hurdle for a user who is familiar with Teradata but not SAS.

The purpose of this document is to describe how a Teradata user can fully leverage the breadth of SAS functionality, including analytics, using only Teradata SQL. No additional products or software are required to enable this functionality. Although the original configuration requires a database administrator's knowledge, the end user is only exposed to Teradata SQL. A Teradata user can then utilize the entire range of benefits of the SAS and Teradata partnership, including performance benefits and transparency.

Integrating SAS® Stored Processes with Teradata Java External Stored Procedures

A SAS Stored Process is packaged SAS code that allows users access to all SAS functionality. The Teradata Java External Stored Procedure (XSP) API enables Teradata users to execute procedures outside of Teradata. Connecting these two technologies enables a Teradata user to access all SAS functionality within a Teradata session and database. The executed SAS Stored Process also has access to all of the data stored on the Teradata server by calling the SAS/ACCESS to Teradata engine. This creates a unified and integrated environment for data processing. The following outline summarizes the steps required to integrate SAS Stored Processes with Teradata Java XSPs:

1. Configure a deployed SAS system to support stored process execution using Trusted Peer connection logic. See the "SAS Configuration" and "Security" sections for details.
2. Download the SAS JAR files that contain supporting Java classes.
3. Write Teradata SQL that defines a Java XSP as described in the "Teradata Configuration" section of this paper.

Figure 1 illustrates the general execution flow involved in integrating SAS Stored Processes and Teradata Java XSPs.

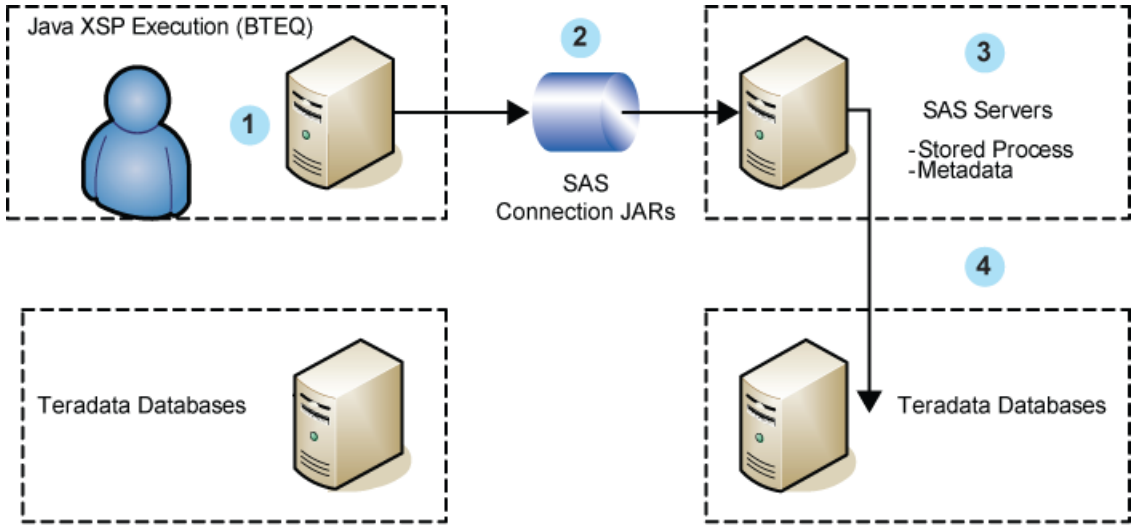


Figure 1: Java External Stored Procedures and SAS Stored Process Connection

The current integration requires SAS 9.2 and Teradata V12 on Linux. Some security models require Teradata V13. Other hosts and versions can probably be supported with minor changes.

SAS® Configuration

The first step is configuring SAS for access by Teradata Java XSPs. A SAS BI deployment is required. The BI system must be installed with SAS Foundation components such as a SAS® Metadata Server and a SAS object spawner. The SAS Metadata Server must be configured to support SAS® Stored Process Servers as well. SAS installation documentation provides the details of this process.

The SAS Metadata Server and object spawner can be installed as system services to avoid having to manually restart the servers on system start-up. If this is not done as part of the initial installation, it can be configured on Windows using the Windows Services control panel or on UNIX using the /etc/rc* startup scripts. The desired result is to have a properly configured SAS Metadata Server and object spawner running before the Teradata Java XSP is defined.

A SAS Stored Process must be defined in the SAS metadata. This definition includes the location of the physical file containing the stored process code as well as the SAS Stored Process Server on which to execute the code. SAS® Management Console can be used for this purpose[1]. Because the stored process can contain any SAS statements that the user wants, the user can invoke the SAS/ACCESS to Teradata engine[2] for access to Teradata data, as well as execute any other SAS functionality.

[1] See the *SAS® 9.2 Stored Processes Developer's Guide* at support.sas.com/documentation/cdl/en/stpug/61271/HTML/default/titlepage.htm for information about creating SAS Stored Processes.

[2] See the Teradata section of the *SAS/ACCESS® 9.2 for Relational Databases: Reference* at support.sas.com/documentation/cdl/en/acreldb/61890/HTML/default/a002299548.htm.

A complete description of SAS Management Console, SAS Stored Process definitions, and SAS/ACCESS engines can be found in other SAS documentation.

Teradata Configuration

Teradata users use the Java External Stored Procedure API to define an entry method using Java source. The user then maps the signature of the entry method into Teradata types and defines the Java XSP accordingly. The Java source is compiled as part of the Java XSP definition process, and the procedure becomes callable from within Teradata.

The following steps show how Teradata SQL is used to define and execute a Java XSP in a manner consistent with the requirements of the SAS Stored Process and Teradata stored procedure integration. It assumes that the user has been defined to Teradata and has been granted the necessary privileges using Teradata semantics as described in Appendix A.

1. Create a Java class named “shell” with a method named “main” (a sample “shell” class can be found with the SAS support JARs named STPrun_logi).

2. This class is then packaged into the shell.jar file in the /tmp directory. Assume that the “shell” class needs to use some classes that are packaged into the support.jar file, which is also in the /tmp directory. In order to call the “main” method in the “shell” class, Teradata must be made aware of these two JARs. Since the Java XSP user is within a BTEQ[3] or other Teradata UI session, special SQL statements can be used to load foreign Java classes.

```
call sqlj.install_jar('sj!/tmp/shell.jar','shell',0);
call sqlj.install_jar('sj!/tmp/support.jar','support',0);
```

3. Execute the following SQL statement to provide the code in shell.jar with visibility into the support.jar file so it can load all the support classes it needs. This is equivalent to modifying the CLASSPATH that is used by code in shell.jar.

```
call sqlj.alter_java_path('shell','(*,support)');
```

4. Define a Teradata Java XSP to call the “main” method in the “shell” class.

```
replace procedure testshell
(inout args varchar(32))
language java
no sql
parameter style java
external name 'shell:shell.main';
```

5. Call the method to execute the Java XSP.

```
call testshell(‘test’);
```

[3] BTEQ is a Teradata command-based user interface that can be used to submit Teradata SQL interactively or in batch.

Security

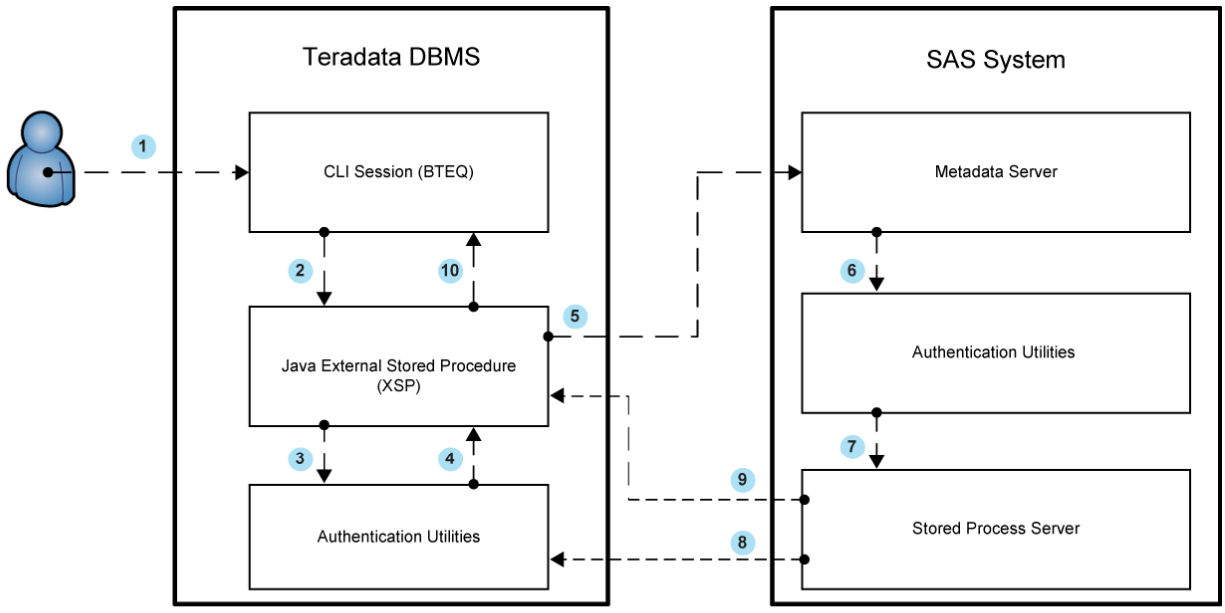
There are two points in the processing of integrated SAS Stored Process and Teradata Java XSP requests at which it is necessary to examine user credentials. First, a Teradata user must be authenticated (validated) by SAS when a connection is made to the SAS servers. Second, the user must be re-authenticated by Teradata if a SAS Stored Process attempts to access Teradata data.

In order to satisfy security requirements while connecting to a SAS server, the Teradata username and password must exist in the SAS metadata and have the appropriate permission, or the server must allow Trusted Peer connections. Because there might be numerous Teradata IDs and passwords, adding them all to SAS metadata might be an onerous task. Using the Trusted Peer security model provides an efficient way to satisfy security requirements.

The Trusted Peer security model is a mechanism in which the SAS system administrator provides a file that describes a connection type. The description includes the client type (for example, Java or C++) and the originating host name and IP address. Connection requests of the type found in the file are then allowed access to the SAS servers.

Trusted Peer security supports several schemes that implement secure access to the Teradata system.

Figure 2 illustrates the SAS single-password store scheme.

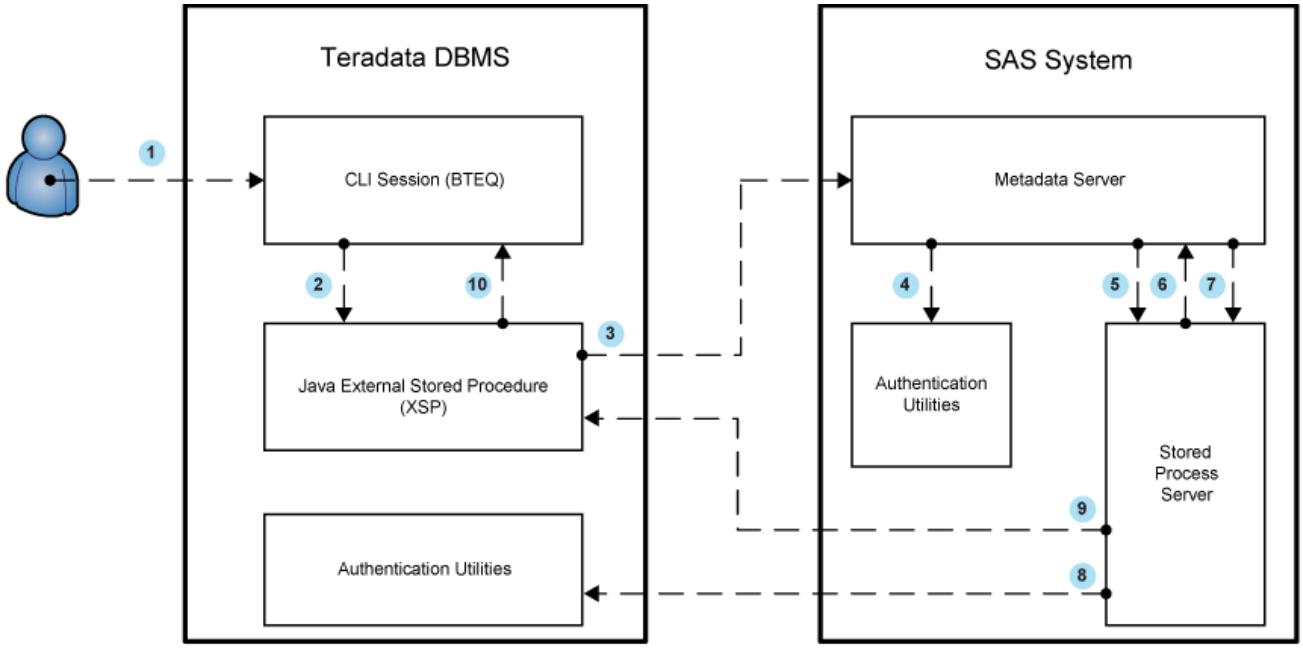


1. User “Fred” logs on to a Teradata session with password “Wilma”.
2. User “Fred” executes Java XSP “SASSTP”.
3. Java XSP requests session username and encrypted password from Teradata authentication utilities.
4. Authentication utility returns user=”Fred” pw=”{001}Barney” (encrypted, one time use password) to the XSP.
5. XSP requests to run SAS stored process “Score” with parameters “Fred” and “{001}Barney”.
6. Metadata server authenticates XSP connection request with Trusted Peer logic.
7. Metadata server executes stored process “Score” with parameters “Fred” and “{001}Barney”.
8. Stored process “Score” executes SAS/ACCESS to Teradata LIBNAME statement with credentials “Fred” and “{001}Barney”.
9. Output parameters and return code for “Score” are sent to the XSP by the stored process server.
10. Output parameters and return code for the XSP are sent to the Teradata session.

Figure 2: SAS Security Scheme – Single-Password Store

In this SAS single-password store scheme, a security token representing the user of the Teradata session is captured and sent to the SAS system as part of the execution request. The contents of this token are assigned as the user credentials if a request to access Teradata data is made from within the SAS Stored Process. In this way, the session user's credentials can be honored when the SAS Stored Process attempts to manipulate data via the SAS/ACCESS to Teradata engine. Support for this scheme is not available at this time and would require changes from Teradata.

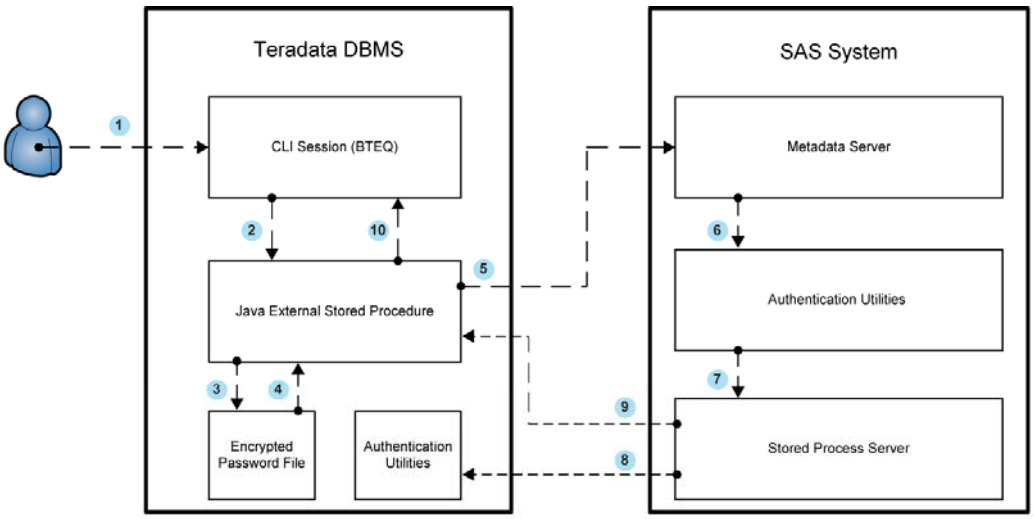
In the SAS multiple-password store scheme, the Teradata session credentials are stored in SAS metadata and acquired by the Java XSPs as illustrated in Figure 3.



1. User "Fred" logs on to a Teradata session with password "Wilma".
2. User "Fred" executes Java XSP "SASSTP".
3. XSP requests to run SAS stored process "Score" with parameter "Fred".
4. Metadata server authenticates XSP connection request with Trusted Peer logic.
5. Metadata server executes stored process "Score" with parameter "Fred".
6. Stored process "Score" requests Teradata password for "Fred" from metadata.
7. Metadata server returns password "Wilma".
8. Stored process "Score" executes SAS/ACCESS to Teradata LIBNAME statement with credentials "Fred" and "Wilma".
9. Output parameters and return code for "Score" are sent to the XSP by the stored process server.
10. Output parameters and return code for the XSP are sent to the Teradata session.

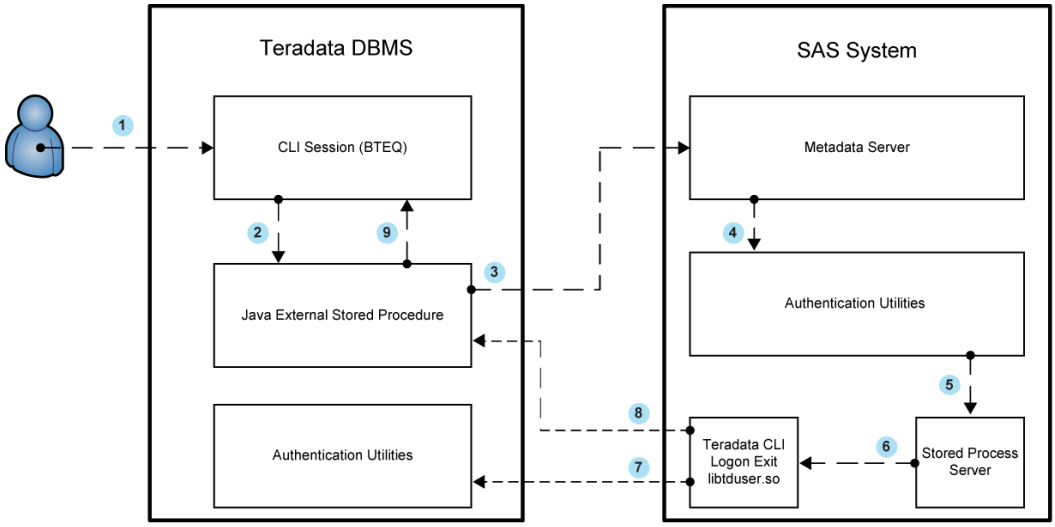
Figure 3: SAS Security Scheme – Multiple-Password Store

Figures 4 and 5 depict schemes in which the session credentials are stored on a Teradata server or a SAS server and acquired via Teradata service routines.



1. User "Fred" logs on to a Teradata session with password "Wilma".
2. User "Fred" executes Java XSP "SASSTP".
3. Java XSP requests encrypted password for "Fred" from Teradata password file (private key encrypted, key known to SAS).
4. Password "Betty" for user "Fred" returned to the XSP.
5. XSP requests to run SAS stored process "Score" with parameters "Fred" and "Betty".
6. Metadata server authenticates XSP connection request with Trusted Peer logic.
7. SAS authentication services converts encrypted password "Betty" to "Wilma" with known private key and executes stored process "Score" with parameters Fred and Betty.
8. Stored process "Score" executes SAS/ACCESS to Teradata LIBNAME statement with credentials Fred and Wilma.
9. Output parameters and return code for "Score" are sent to the XSP by the stored process server.
10. Output parameters and return code for the XSP are sent to the Teradata session.

Figure 4: Teradata Security Scheme – Passwords Stored on a Teradata Server



1. User "Fred" logs on to a Teradata session with password "Wilma".
2. User "Fred" executes Java XSP "SASSTP".
3. XSP requests to run SAS stored process "Score" with parameter "Fred".
4. Metadata server authenticates XSP connection request with Trusted Peer logic.
5. Metadata server executes stored process "Score" with parameter "Fred".
6. Stored process "Score" executes SAS/ACCESS to Teradata LIBNAME statement with user "Fred" and blank password, invoking Teradata logon exit routine.
7. Teradata logon exit routine converts Fred's blank password to "Wilma" via predefined encrypted file and validates it.
8. Output parameters and return code for "Score" are sent to the XSP by the stored process server.
9. Output parameters and return code for the XSP are sent to the Teradata session.

Figure 5: Teradata Security Scheme – Passwords Stored on a SAS Server

Note: Query banding and the other Teradata schemes are available in Teradata V13. They are not supported using Teradata V12 or earlier.

Example: Creating and Executing a SAS® Stored Process from within Teradata

This example shows the steps necessary to configure an integrated Teradata Java XSP with a SAS Stored Process. It assumes that the SAS servers are running and that the SAS system has been configured with the Trusted Peer logic described previously and includes a SAS Stored Process named testtokentime. This example also assumes that the Teradata session user has the appropriate disk space and permissions to create Teradata stored procedures.

1. Copy the necessary supporting JAR files, which are supplied by SAS, to an accessible directory on the Teradata server (for example, the /tmp directory is used in this example). Note that some of the JARs included in this example are used to extend SAS functionality and are not required for this example. They are included to provide a complete list of available SAS JARs.

2. Use the following INSTALL_JAR SQL commands to give the Teradata class loader access to the JARs and their associated classes. The STPrun_logi JAR contains the Java code that bridges between the Teradata SQL calls and the SAS Stored Process Server calls, including parameter marshaling and executing the SAS Stored Process. The rest of the JARs are used by STPrun_logi to execute the stored process. They are not used directly by Teradata[4].

```
call sqlj.install_jar('sj!/tmp/stprun_logi.jar','sasstprun_logi',0);
call sqlj.install_jar('sj!/tmp/activation.jar','sasact',0);
call sqlj.install_jar('sj!/tmp/antlr-2.7.7.jar','sasant2',0);
call sqlj.install_jar('sj!/tmp/antlr-3.0.1.jar','sasant3',0);
call sqlj.install_jar('sj!/tmp/antlr.jar','sasant',0);
call sqlj.install_jar('sj!/tmp/antlr-runtime-3.0.1.jar','sasatr',0);
call sqlj.install_jar('sj!/tmp/asm-attrs.jar','sasasma',0);
call sqlj.install_jar('sj!/tmp/asm.jar','sasasm',0);
call sqlj.install_jar('sj!/tmp/c3p0.jar','sasc3p0',0);
call sqlj.install_jar('sj!/tmp/cglib.jar','sascgl',0);
call sqlj.install_jar('sj!/tmp/commons-cli.jar','sasccli',0);
call sqlj.install_jar('sj!/tmp/commons-codec.jar','sasccod',0);
call sqlj.install_jar('sj!/tmp/commons-collections.jar','sasccol',0);
call sqlj.install_jar('sj!/tmp/commons-httpclient.jar','saschttp',0);
call sqlj.install_jar('sj!/tmp/commons-lang.jar','sasclng',0);
call sqlj.install_jar('sj!/tmp/commons-logging.jar','sasclog',0);
call sqlj.install_jar('sj!/tmp/connector.jar','sascon',0);
call sqlj.install_jar('sj!/tmp/dl.util.concurrent.jar','sasdl',0);
call sqlj.install_jar('sj!/tmp/dom4j.jar','sasdom',0);
call sqlj.install_jar('sj!/tmp/ehcache.jar','sasehc',0);
call sqlj.install_jar('sj!/tmp/firebirdsql-full.jar','sasfir',0);
call sqlj.install_jar('sj!/tmp/hibernate3.jar','sashib3',0);
call sqlj.install_jar('sj!/tmp/hibernate-tools.jar','sashib',0);
call sqlj.install_jar('sj!/tmp/icu4j.jar','sasicu',0);
call sqlj.install_jar('sj!/tmp/imq.jar','sasimq',0);
call sqlj.install_jar('sj!/tmp/jdom.jar','sasjdom',0);
call sqlj.install_jar('sj!/tmp/jhall.jar','sasjhal',0);
call sqlj.install_jar('sj!/tmp/jh.jar','sasjh',0);
call sqlj.install_jar('sj!/tmp/jms.jar','sasjms',0);
call sqlj.install_jar('sj!/tmp/jsp-api.jar','sasjsp',0);
call sqlj.install_jar('sj!/tmp/jta.jar','sasjta',0);
call sqlj.install_jar('sj!/tmp/junit.jar','sasjuni',0);
```

[4] The size of some of the supporting SAS JARs exceeded Teradata limits, so they were divided into sub-JARs.

```
call sqlj.install_jar('sj!/tmp/junitperf.jar','sasjnuip',0);
call sqlj.install_jar('sj!/tmp/log4j.jar','saslog4',0);
call sqlj.install_jar('sj!/tmp/mail.jar','sasmail',0);
call sqlj.install_jar('sj!/tmp/mxlangman.jar','sasmxl',0);
call sqlj.install_jar('sj!/tmp/platformservicesdeployment.jar','saspsd',0);
call sqlj.install_jar('sj!/tmp/saaj-api.jar','sassapi',0);
call sqlj.install_jar('sj!/tmp/saaj-impl.jar','sassimp',0);
call sqlj.install_jar('sj!/tmp/sas.antlr.jar','sassant',0);
call sqlj.install_jar('sj!/tmp/sas.arm.log4j.jar','sassarm',0);
call sqlj.install_jar('sj!/tmp/sas.core.jar','sasscor',0);
call sqlj.install_jar('sj!/tmp/sas.core.net.jar','sasscorn',0);
call sqlj.install_jar('sj!/tmp/sas.core.setinit.jar','sasscors',0);
call sqlj.install_jar('sj!/tmp/sas.entities.jar','sassej',0);
call sqlj.install_jar('sj!/tmp/sas.entities.util.jar','sasseu',0);
call sqlj.install_jar('sj!/tmp/sas.expr.visuals.jar','sassev',0);
call sqlj.install_jar('sj!/tmp/sas.icons.jar','sassico',0);
call sqlj.install_jar('sj!/tmp/sas.iom.prx.sasmdx.jar','saspmdx',0);
call sqlj.install_jar('sj!/tmp/sas.iquery.services.jar','sasiq',0);
call sqlj.install_jar('sj!/tmp/sas.iqueryutil.jar','sasiqu',0);
call sqlj.install_jar('sj!/tmp/sas.nls.collator.jar','sassnls',0);
call sqlj.install_jar('sj!/tmp/sas.oma.joma.jar','sassoj',0);
call sqlj.install_jar('sj!/tmp/sas.oma.joma.rmt1.jar','sasomr1',0);
call sqlj.install_jar('sj!/tmp/sas.oma.joma.rmt2.jar','sasomr2',0);
call sqlj.install_jar('sj!/tmp/sas.oma.joma.rmt3.jar','sasomr3',0);
call sqlj.install_jar('sj!/tmp/sas.oma.omi.jar','sassoo',0);
call sqlj.install_jar('sj!/tmp/sas.omi.checkproxy.jar','sassoch',0);
call sqlj.install_jar('sj!/tmp/sas.omi.permissions.jar','sassop',0);
call sqlj.install_jar('sj!/tmp/sas.omi.util.jar','sassou',0);
call sqlj.install_jar('sj!/tmp/sas.report.config.jar','sassrcfg',0);
call sqlj.install_jar('sj!/tmp/sas.rpf.jar','sassrp',0);
call sqlj.install_jar('sj!/tmp/sas.storage.jar','sasssto',0);
call sqlj.install_jar('sj!/tmp/sas.svc.cache.jar','sasssca',0);
call sqlj.install_jar('sj!/tmp/sas.svc.connection.jar','sasssco',0);
call sqlj.install_jar('sj!/tmp/sas.svc.connection.platform.jar','sassscp',0);
call sqlj.install_jar('sj!/tmp/sas.svc.core1.jar','sascor1',0);
call sqlj.install_jar('sj!/tmp/sas.svc.core2.jar','sascor2',0);
call sqlj.install_jar('sj!/tmp/sas.svc.events.jar','sassse',0);
call sqlj.install_jar('sj!/tmp/sas.svc.events.samples.jar','sassses',0);
call sqlj.install_jar('sj!/tmp/sas.svc.hibernate.jar','sasssh',0);
call sqlj.install_jar('sj!/tmp/sas.svc.publish.jar','sasssp',0);
call sqlj.install_jar('sj!/tmp/sas.svc.sec.login.jar','sassscl',0);
call sqlj.install_jar('sj!/tmp/sas.svc.storedprocess.jar','sassss',0);
call sqlj.install_jar('sj!/tmp/sas.svc.webdav.jar','sasssw',0);
call sqlj.install_jar('sj!/tmp/sas.swing.jar','sassswi',0);
call sqlj.install_jar('sj!/tmp/sas.swing.remote.jar','sasssr',0);
call sqlj.install_jar('sj!/tmp/sas.test.junit.jar','sasstj',0);
call sqlj.install_jar('sj!/tmp/sas.themeresources.tools.jar','sasstt',0);
call sqlj.install_jar('sj!/tmp/sas.tools.modelcompiler.jar','sasstm',0);
call sqlj.install_jar('sj!/tmp/security.jar','sassec',0);
call sqlj.install_jar('sj!/tmp/servlet-api.jar','sasser',0);
call sqlj.install_jar('sj!/tmp/slide-webdavlib.jar','sasslid',0);
call sqlj.install_jar('sj!/tmp/src.jar','sassrc',0);
call sqlj.install_jar('sj!/tmp/stringtemplate-3.1b1.jar','sasstr',0);
call sqlj.install_jar('sj!/tmp/tidy.jar','sastidy',0);
```

3. Use the ALTER_JAVA_PATH SQL command to make all the SAS JARs aware of each other in the Teradata session and to provide the SASSTPrun_logi JAR with the visibility to the SAS middle-tier classes. This command performs a similar function to altering the Java CLASSPATH variable; think of this step as putting all these JARs into the Java CLASSPATH that is used when running code in the SASSTPrun_logi JAR.

```
call sqlj.alter_java_path('sasstprun_logi','(*,sasact)(*,sasant2)(*,sasant
3)(*,sasant)(*,sasantr)(*,sasasma)(*,sasasm)(*,sasc3p0)(*,sascgl)(*,sasccl
i)(*,sasccod)(*,sasccol)(*,saschttp)(*,sasclng)(*,sasclog)(*,sascon)(*,sas
dl)(*,sasdom)(*,sasehc)(*,sasfir)(*,sashel)(*,sashib3)(*,sashib)(*,sasicu)
(*,sasimq)(*,sasjdom)(*,sasjhal)(*,sasjh)(*,sasjms)(*,sasjsp)(*,sasjta)(*,
sasjuni)(*,sasjnuip)(*,saslog4)(*,sasmail)(*,sasmxl)(*,saspsd)(*,sassapi)(
*,sassimp)(*,sassant)(*,sassarm)(*,sasscor)(*,sasscorn)(*,sasscors)(*,sass
ej)(*,sasseu)(*,sassev)(*,sassico)(*,saspmdx)(*,sasiq)(*,sasiqu)(*,sassnls
)(*,sassoj)(*,sasomr1)(*,sasomr2)(*,sasomr3)(*,sassoo)(*,sassoch)(*,sassop
)(*,sassou)(*,sassrcfg)(*,sassrp)(*,sasssto)(*,sasssca)(*,sasssco)(*,sasss
cp)(*,sascor1)(*,sascor2)(*,sassse)(*,sassses)(*,sasssh)(*,sasssp)(*,sasss
cl)(*,sassss)(*,sasssw)(*,sassswi)(*,sasssr)(*,sasstj)(*,sasstt)(*,sasstm)
(*,sassec)(*,sasser)(*,sasslid)(*,sassrc)(*,sasstr)(*,sastestfile)(*,saste
stp2)(*,sastestp)(*,sastidy)(*,sastoken)');
```

4. Create a SAS Stored Process. The following code creates a SAS Stored Process named testtokentime. This stored process calls back into the Teradata system and manipulates data via the SAS/ACCESS to Teradata LIBNAME engine, although any licensed SAS code will execute as expected. The stored process updates the table xspschema.testme with the current time in the second column.

```
   /* Identifying header for a stored process */

*ProcessBody;

   /* Call the SAS/ACCESS to Teradata engine using the SAS LIBNAME
      statement */

libname a teradata user=username pass=userpass schema=xspschema;

   /* Remove the Teradata data table */

proc delete data=a.testme;run;

   /* Create a new table */

data a.testme;

col1=1;col2='test1              ';output;
col1=2;col2='test2              ';output;
col1=3;col2='test3              ';output;

run;

   /* Update a field in the table using SQL */

proc sql;
update a.testme set col2=put(datetime(), datetime18.) where col1=2; quit;

run;
```

5. Define the SAS Stored Process in the SAS metadata using SAS Management Console. The SAS Stored Process documentation provides details of this operation.

6. Define and call the Teradata Java XSP using the following Teradata SQL. The Java XSP calls the main entry point in the SASSTPrun_logi JAR. The SASSTPrun_logi JAR contains a Java class named STPrun_logi, which contains a method named SASStoredProcess. This method takes SAS Metadata Server connection information and the name of the SAS Stored Process stored in the SAS Metadata Server. It then executes that stored process. After the Java XSP is created, it is called with the arguments that identify the SAS Metadata Server and the testtokentime stored process. This same Java XSP can be used to execute any SAS Stored Process defined in any SAS Metadata Server[5].

```
   /* define the java xsp */
replace procedure sasstoredprocess
(in  metahost  varchar(32),
 in  metaport  varchar(32),
 in  stpname   varchar(32),
 out rc        integer)
language java
no sql
parameter style java
external name 'sasstprun_logi:stprun_logi.sasstoredprocess';

   /* execute the java xsp and the stored process 'testtokentime'
      by proxy */
call sasstoredprocess('sasmetadatahosthostname', '8561', 'testtokentime', ?);
```

Leveraging the Power of SAS® Analytics

In the following examples, SAS Stored Processes perform complex SAS Analytics and then communicate with the Teradata Java XSP through output parameters.

These are only examples to show how Teradata users can take advantage of the powerful analytics capability encapsulated in SAS Stored Processes using only their knowledge of Teradata syntax. These examples run inside the Teradata database. Although these examples cannot be used verbatim, they can be modified to work on a properly configured system.

Example 1: Create and Print Customer Information

This SAS code uses Teradata SQL and the SAS FREQ procedure (PROC FREQ) to illustrate using Teradata user-defined functions (UDFs) with SAS formats. First a table is created in the Teradata database using SAS explicit pass-through technology. Then the table is populated with customer information about the requested branch. Finally a frequency report on the customer information is generated.

```
   /* Input parameters  */

%let tablename=&stpInValue;
%let branchId=&stpInValue2;

   /* Specify a convenient location for the log file using the SAS PRINTTO
      procedure. */
```

[5] The parameterization of the final procedure is optional. Top-level shells (such as STPrun_logi) can be provided that hide the machine information and the procedure name. It might be desirable to provide more detailed parameters to the stored process than simply an "out" variable, as shown.

```
proc printto log="&hmeqdir./results/&tablename..log";
run;

   /* Build the WHERE clause subsetting the data to only data for the branch id that
is passed in. */

%let branchClause=b.branchId=&branchId. and;

   /* Create a SAS macro variable to control SQL generation for PROC FREQ:
      Yes - SQL generation for PROC FREQ
      No  - Read each record into SAS and process */

%let SYS_FREQSQL=YES;

   /* Make it chatty so we can see what Freq SQL generation is doing. */

%let SYS_FREQSQLDEBUG=YES;  /* Yes or No */

   /* Turn on timing statistics and set output width. */

options fullstimer;
options ls=90;

%let outTable=&ipusrlib..&tablename.;

%let tempTablename=tmp_&tablename;
%let tempTable=&ipusrlib..&tempTablename.;

%cleanTable(&tempTable.);

   /* Show SAS SQL procedure explicit pass-through to connect to Teradata and
      create a table:
         - uses SAS UDFs
         - leverages join index in TD */

proc sql noprint;
  connect to teradata (user=&dbuser. pw=&dbpw. server=&dbserver. mode=teradata);

  execute (
        create table &tempTablename. as  (
        select
               b.branchId,
               sas_udf_putc(c.state,'$abbr2region.') as region,
               a.loanAmount as loanAmount,
               c.jobTitle as jobTitle
        from &database..branches as b,
             &database..customer as c,
             &database..applicationCustomerBranch as acb,
             &database..applications as a
        where &branchClause. b.branchId=acb.branchId
              and acb.customerId=c.customerId
              and acb.loanId=a.loanId
         ) with data;
  ) by teradata;
quit;

run;
```

```
   /* Execute PROC FREQ on the table using pass-through and print the log. */

%cleanTable(&outTable.);
proc freq data=&tempTable. noprint;
     format loanAmount dollar12.2;
     table region*jobTitle / out=&outTable.(rename=(count=loanAmount));
     weight loanAmount;
run;

proc printto log=log;
run;
```

Example 2: Graphs and Loan Plotting

The following example uses the SAS Output Delivery System (ODS) for reporting. It identifies the branch with the worst-performing loans.

```
   /* Specify an output location. */

%let outlib=work;

   /* Include branch loan forecast. */

%inc "SASEnvironment/SASCode/Jobs/branchLoanForecast.sas";

ods pdf file="Data/hmeq/results/&tablename..pdf";
ods listing close;
     title1 "Branch Loan Performance";

    axis2 label=(a=-90 r=90 "Branches" );
    symbol1 v = dot    i = join l = 1;
    symbol2 v = star   i = join l = 2;
    symbol3 v = circle i = join l = 2;

    proc gplot data=&outlib..&tablename.;
       by branchId;
       plot loanAmount  * month = 1 /
       href="&sysdate."d
;
    run;
quit;
ods pdf close;
ods listing;

%let stpOutRc=0;
```

These examples reflect using SAS Stored Processes to leverage functionality such as SAS/ACCESS engine technology, SAS formats, and advanced analytics. The SAS Stored Processes are executed inside a Teradata database using Teradata SQL. For more information about deploying SAS formats in Teradata, see “Deploying and Using SAS Formats in Teradata” in the *SAS/ACCESS for Relational Databases: Reference*, and refer to other SAS/ACCESS documentation.

Appendix A: Creating and Configuring Teradata Java XSPs to Invoke SAS® Stored Processes

This document describes how to install and configure Java XSPs within the Teradata database to invoke SAS Stored Processes.

Grant Appropriate Authorization

Authorization to create Java XSPs within the Teradata database is needed. Teradata provides a database (SQLJ) with the appropriate procedures and calls for Java XSP creation.

To grant the logon ID sas_proc the permission to create these procedures, connect to Teradata as DBC and issue the following SQL commands:

```
grant create procedure on <database> to sas_proc with grant option;
grant drop procedure on <database> to sas_proc with grant option;
grant execute procedure on <database> to sas_proc with grant option;
grant create function  on <database> to sas_proc with grant option;
grant create external procedure on <database> to sas_proc with grant option;
grant alter function on <database> to sas_proc;
grant alter procedure on <database> to sas_proc;
grant alter external procedure on <database> to sas_proc with grant option;
```

Install SAS JAR Files

To establish connectivity to the SAS servers, instantiate the necessary SAS JAR files into the Teradata database by placing the appropriate files into the C:\ SAS_JARS directory on the database as shown in the following example script. Connect to Teradata as DBC. In the script, CJ! = CLIENT COMPUTER JAVA, followed by a directory.

```
call sqlj.install_jar('cj!c:\ sas_jars\sas.core.jar','jarxspcore',0);
call sqlj.install_jar('cj!c:\ sas_jars \sas.iom.tools.jar','jarxsptools',0)
call sqlj.install_jar('cj!c:\ sas_jars \sas.oma.omi.jar','jarxspomi',0);
call sqlj.install_jar('cj!c:\ sas_jars \sas.svc.connection.jar','jarxspcnct',0);
call sqlj.install_jar('cj!c:\ sas_jars \sas.svc.storedprocess.jar','jarxspstp',0);
call sqlj.install_jar('cj!c:\ sas_jars \streamreader.jar','jarxspstream',0);
```

The Java XSP can now be created. See Appendix B for the sample Java program sas_call.

Create and Configure the Java XSP

To create the appropriate class and JAR files, perform the following steps:

1. Set JAVA_HOME in your environment to JDK1.5.0_15:

   ```
   set java_home=c:\program files\java\jkd1.5.0_15
   ```

2. Set your JAR_ROOT to include the location of the client side JAR files placed into the database.

   ```
   set jar_root=c:\sas_jars
   ```

3. Add the JAR files to your classpath.

```
set classpath=%classpath%;.;%jar_root%\sas.svc.connection.jar;%jar_root
%\sas.svc.storedprocess.jar;%jar_root%\sas.core.jar;%jar_root%\sas.iom.
tools.jar;%jar_root%\sas.oma.omi.jar
```

4. Compile the sas_call program and create a JAR file out of the CLASS file.

```
"%java_home%"\bin\javac -target 1.5 -g sas_call.java
"%java_home%"\bin\jar -cvf sas_call.jar sas_call.class
```

5. Create the stored procedure. From a BTEQ window, connect to Teradata as sas_proc.

```
login tdserver,sas_proc
```

6. Make sure the sqltest procedure doesn't already exist.

```
drop procedure sqltest;
```

7. Make sure the sas_call JAR file doesn't exist.

```
call sqlj.remove_jar('sas_call',0);
```

8. Install the sas_call JAR file you created.

```
call
sqlj.install_jar('cj!c:\sas\projects\storedprocs\indb\sas_call.jar',
'sas_call', 0);
```

9. Set the Java path for sas_call *within* the Teradata environment.

```
call sqlj.alter_java_path('sas_call','(*,jarxspcore)(*,jarxsptools)(*,
jarxspomi)(*,jarxspcnct)(*,jarxspstp)(*,jarxspstream)');
```

10. Create the sqltest Java stored procedure.

```
create procedure sqltest
       ( inout r varchar(50))
       language java modifies sql data
       parameter style java
       external name 'sas_call:sas_call.main';
```

11. Test the procedure

```
call sqltest('foo');
```

12. Verify the program executed successfully by looking at the SAS logs.

Appendix B: SAS_CALL Sample Java Program

```
/*****************************************************************************
    NAME:   sas_call.java
    PURPOSE: Sample Teradata Java XSP that calls a SAS Stored Process

*****************************************************************************/

import java.sql.*;
import java.io.*;

import java.net.URLEncoder;
import java.util.Properties;
import java.util.Enumeration;
import java.lang.reflect.*;
//import java.io.FileOutputStream;
//import java.io.IOException;
import java.sql.SQLException;

/*************************************************************
    Class:    sas_call
    Description: Class Definition
*************************************************************/
public class sas_call
{

    // Variable initialization for LOGGING JXSP
    private static String LOG_FILE = "/usr/Sas_procs/STPrun_SAS.log";
    private static FileOutputStream log_stream;

    /*************************************************************
          Function:    main
          Description: Entry point into .java program
    *************************************************************/
    public static void main(String argv[])   {

          // Initialize Variables for SAS Metadata and Stored Process Servers
          String sMetaHost = "";
          String sMetaPort = "";
          String sMetaUser = "";
          String sMetaPass = "";
          String sSTPHost = "";
          String sSTPPort = "";
          String sSTPUser = "";
          String sSTPPass = "";

          // Start Logging and write initial log message
          log_start();
          log_msg("sas_call Entry Point");
          String[] aServerInfo = new String[4];
          Try

          {
                GetSASServerInfo("METADATA", aServerInfo);
          }
          catch (Exception e)
```

```
{
      e.printStackTrace();
}
sMetaHost = aServerInfo[0];
sMetaPort = aServerInfo[1];
sMetaUser = aServerInfo[2];
sMetaPass = aServerInfo[3];
try
{
      GetSASServerInfo("STOREDPROC", aServerInfo);
}
catch (Exception e)
{
      e.printStackTrace();
}
sSTPHost = aServerInfo[0];
sSTPPort = aServerInfo[1];
sSTPUser = aServerInfo[2];
sSTPPass = aServerInfo[3];

// Define Instance of SAS Stored Process Server
com.sas.iom.SASStoredProcess.IStoredProcessServer stpServer;
com.sas.iom.SASStoredProcess.IStoredProcessContext stpContext;

// Define location of Stored Process to run AND the Stored Process Program
java.lang.String repository = "c:\\SAS\\Projects\\StoredProcs";
java.lang.String storedProc = "TDSample.sas";

java.lang.String[] Names = new String[1];
java.lang.String[] Values = new String[1];

// Create the SAS Log Object
org.omg.CORBA.StringHolder formattedLog = new
 org.omg.CORBA.StringHolder();

java.lang.String sessionID = "";

// Create the UUID
org.omg.CORBA.StringHolder uuid = new org.omg.CORBA.StringHolder();

int programStatus;

// Set up canned Name/Value pairs for defining macros.
Names[0] = "_PROGRAM";
Values[0] = "storedProc";

// Create a Properties object to use as a Server Definition
java.util.Properties prop = new java.util.Properties();

prop.put("host", sSTPHost);
prop.put("port", sSTPPort);
prop.put("userName", sSTPUser);
prop.put("password", sSTPPass);
prop.put("encryptionPolicy", "required");
prop.put("encryptionAlgorithms", "sasproprietary");
prop.put("encryptionContent", "all");
```

```
        // Create the ORB
        org.omg.CORBA.ORB orb = new com.sas.net.brg.orb.BrgOrb();

        // Create the URL Property
        String url = propertiesToUrl(prop);

        // Create the Orb Object
        org.omg.CORBA.Object obj = orb.string_to_object(url);

        // Create an instance of the SAS Stored Process Server
           stpServer = com.sas.iom.SASStoredProcess.
                       IStoredProcessServerHelper.narrow(obj);

        try
        {

               // Create a Stored Process Server Context
               stpContext = stpServer.CreateContext(repository, storedProc, Names,
                            Values, sessionID);
               // Run the Stored Process
               stpContext.Execute(uuid);

               programStatus = stpContext.SASConditionCode();

               // Retrieve the SAS Log
               stpContext.GetFormattedSasLog(stpContext.LOG_FORMAT_HTML, 500,
              formattedLog);

                      log_msg("SAS Log");
                      log_msg(formattedLog.value);

               stpContext.Close();
        }

        catch (com.sas.iom.SASStoredProcess.IStoredProcessServerPackage.
               BadSessionID bid)
        {
               bid.printStackTrace();
        }
        catch (com.sas.iom.SASIOMDefs.GenericError gen)
        {
               gen.printStackTrace();
        }
        catch (Exception e)
        {
               e.printStackTrace();
        }
    } // END main
```

```
/*************************************************************************
       Function:    GetSASServerInfo
       Description: Read server information from the Teradata Database
*************************************************************************/

private static void GetSASServerInfo(String sServerType, String aServerInfo[])
throws SQLException
{
       try
       {
       // Establish a jdbc connection so the .java program can read data from
          a Teradata table
       Connection con = DriverManager.getConnection
                      ( "jdbc:default:connection/LOG=DEBUG" );

              log_msg("con established");

       // Define SQL to execute
       String sSelect =
              "SELECT    server_host_name" +
                           ", server_port" +
                           ", server_user_name" +
                           ", server_user_pass" +
              " FROM     SQLJ.SAS_SERVER_INFO" +
              " WHERE    server_type = '" + sServerType + "'";

       // Execute the SQL statement
       Statement stmt = con.createStatement();

       // Open the resultset
       ResultSet rs = stmt.executeQuery( sSelect );

       // Retrieve the row information that contains the SAS Metadata
       // and Stored Process Servers connection information.
       while (rs.next()) {
              aServerInfo[0] = rs.getString(1);
              aServerInfo[1] = rs.getString(2);
              aServerInfo[2] = rs.getString(3);
              aServerInfo[3] = rs.getString(4);

                     log_msg("SQL: " + sSelect);
                     log_msg("Server Host: " + aServerInfo[0]);
                     log_msg("Server Port: " + aServerInfo[1]);
                     log_msg("Server User: " + aServerInfo[2]);
                     log_msg("Server Pass: " + aServerInfo[3]);
       }

       // Close the SQL Execution and Resultset
       rs.close();
       stmt.close();
       // ALL SAS SERVER INFORMATION IS NOW DEFINED
       }
       catch (Exception e)
       {
              e.printStackTrace();
       }

}      // End GetSASServerInfo
```

```
/************************************************************************
       Function:    propertiesToUrl
       Description: Create a URL String that points to a SAS Stored
              Process Server
************************************************************************/

private static String propertiesToUrl(Properties properties)
{
       StringBuffer buffer = new StringBuffer("bridge://");
       buffer.append(properties.getProperty("host"));
       buffer.append(":");
       buffer.append(properties.getProperty("port"));
       buffer.append("/");
       buffer.append("15931E31-667F-11D5-8804-00C04F35AC8C");
       StringBuffer query = null;

       Enumeration propertyNames = properties.propertyNames();

       while (propertyNames.hasMoreElements())
       {
              String propertyName = (String)propertyNames.nextElement();

              if (!propertyName.equals("host") && !propertyName.equals("port"))
              {
                     if (query == null)
                     {
                            query = new StringBuffer();
                     }
                     else
                     {
                            query.append("&");
                     }

                     // query.append(URLEncoder.encode(propertyName));
                     query.append(propertyName);
                     query.append("=");
                     query.append(properties.getProperty(propertyName));

                     // query.append(URLEncoder.encode
                       (properties.getProperty(propertyName)));
              }
       }

       if (query != null)
       {
              buffer.append("?");
              buffer.append(query.toString());
       }

       buffer.append("&debugLevel=0");

       // debugLevel=X
       //X=0: no trace
       //X=1: handshaking trace only
       //X=2: human readable packet level trace
       //X=3: hex dump of packets sent and received
       return buffer.toString();
}      // END propertiesToUrl
```

```
	/*************************************************************************
		Function:    log_msg
		Description: Write a string log message to the LINUX log file
	*************************************************************************/

	private static void log_msg(String msg)
	{
		String ls = System.getProperty("line.separator");
		try
		{
			System.out.println(msg);
			log_stream.write(new String(msg + ls).getBytes());
			log_stream.flush();
		}
		catch (Exception e)
		{
		}
	}	//END log_msg

	/*************************************************************
		Function:    log_start
		Description:  Open a log file for APPEND
	*************************************************************/

	private static void log_start()
	{
		try
		{
			log_stream = new FileOutputStream(LOG_FILE, true);
		}
		catch (Exception e)
		{
		}
	}	// END log_start

	/*************************************************************
		Function:    log_stop
		Description: Close the log file
	*************************************************************/

	private static void log_stop()
	{
		try
		{
			log_stream.close();
		}
		catch (Exception e)
		{
		}
	}	// END log_stop
}	// END sas_call

	/*************************************************************/
```

Contact Information

Comments and questions are valued and encouraged. Contact the authors:

Tom Bednarek
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Phone: (919) 677-8000 x17755
E-mail: Tom.Bednarek@sas.com

Gary Spakes
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Phone: (919) 677-8000 x15305
E-mail: Gary.Spakes@sas.com