Paper SAS1906-2018

How to Load Relational Data into SAS® Cloud Analytic Services Using Java

Salman Maher SAS Institute Inc., Cary, NC

ABSTRACT

Java is one of the most popular programming languages available today. Did you know that you can easily use Java with SAS® Cloud Analytic Services (CAS)? This presentation shows you how to do this and more. During this presentation, we cover the following tasks:

- how to load the Java libraries required to invoke CAS actions within your Java program
- how to invoke CAS actions from Java
- how to load data from a relational database into CAS

You will walk away from this presentation confident that you can read and write data from CAS to a relational database using a Java program.

INTRODUCTION

The SAS Java Client interface for SAS® Viya® provides a Java interface to interact with SAS Cloud Analytic Services (CAS), the analytics engine for the SAS Viya platform. With this software, you can write Java programs that load data into CAS and invoke CAS actions to transform, summarize, model, and score the data. Classes are provided that correspond to each CAS action, and you can use Java to post-process CAS result tables. This paper will show you how to use the Java interface to connect to relational data sources using a caslib and will discuss some of the CAS actions that can be used with data source caslibs.

GETTING STARTED

To use Java with CAS, you must have a Java 8 run-time environment. SAS recommends that you use the latest version of Java 8 that is available. In addition, the following three JAR files are required:

- SAS Viya for Java JAR file. Download it from https://support.sas.com/downloads/package.htm?pid=1976
- ANTLR JAR. Download it from http://www.antlr.org/download.html or https://mvnrepository.com/artifact/org.antlr/antlr-runtime
- Google Protocol Buffers JAR file. Download it from https://mvnrepository.com/artifact/com.google.protobuf/protobuf-java

The SAS Viya cas-client JAR file provides the Java interface to interact with CAS. In addition to the cas-client JAR file, you will also need to have ANTLR run-time JAR 3.5.2 or later and Google Protocol Buffers JAR 2.6.1 or later. Once you have the Java 8 installed and the three JAR files on your CLASSPATH, you are ready to start.

CONNECTING TO CAS SERVER

Before you can submit any actions to the CAS server, you need to create a CAS client instance. To make a connection, you will need to know the host name and the port number of the CAS server, in addition to your user name and password. The Java code to create your CAS client instance would look like the following example:

```
CASClientInterface client = new CASClient("server-name.mycompany.com", 5570,
"username", " password");
```

You can also supply your credentials by creating an authinfo file. For more information, see Client Authentication Using an Authinfo File in the Getting Stared with SAS Viya for Java documentation. When using an authinfo file, you will only pass in the host and port to the CASClient constructor when you create your CAS client instance. Here is an example:

```
CASClientInterface client = new CASClient("server-name.mycompany.com", 5570)
```

The instance of the CASClient class is used to connect to CAS services and start your session. Once you are connected, all resources are only available to that CASClient instance. Any action that you want to submit to the CAS server is submitted by passing a CAS action option object to the CASClient.invoke method. No action is taken on the CAS server unless it is submitted via the CASClient.invoke method.

CREATING A CASLIB

Now that you have a CASClient instance, you are ready to define your caslib for your data source. Using SAS Studio, you would create your caslib using a caslib statement. For example, your caslib for a PostgreSQL data source will look like this:

```
    /* creates new caslib */
    caslib pgLib sessref=mysess
    datasource=(srctype="postgres",
                server="server",
                database="test",
                username="user",
                password="password"
                );
```

In Java you will create a caslib definition using a specific data source class, such as Dspostgres for PostgreSQL. All the data source classes are part of the com.sas.cas.actions package, with the two-letter prefix “Ds” for data source. To define a data source caslib instance for PostgreSQL like the caslib statement shown above, the Java code would resemble the following:

```
    Dspostgres dsPostgreSQL = new Dspostgres();
    dsPostgreSQL.setServer("server");
    dsPostgreSQL.setUser("user").setPassword("password");
    dsPostgreSQL.setDatabase("test").setSchema("mySchema");
```

Creating the Dspostgres instance does not create the caslib on the CAS server. You must also associate the Dspostgres instance with a caslib action and invoke that action on the CAS client that you created earlier. The Java code to create a caslib named “pgLib” for your PostgreSQL data source instance is as follows:

```
    // Define the casLib for dsPostgreSQL using the AddCaslibOptions class.
    AddCaslibOptions addCasLibOptions = new AddCaslibOptions();
    addCasLibOptions.setName(“pgLib”);
    // assign the data source to the action
    addCasLibOptions.setParameter(AddCaslibOptions.KEY_DATASOURCE,
                                  dsPostgreSQL);
```

The com.sas.cas.actions.table.AddCaslibOptions class identifies the set of caslib action options to invoke. The AddCaslibOptions.setName method specifies the name of your caslib. You will pass this name via a call to setCaslib on all the Table actions that are covered in this paper. The AddCaslibOptions.setParameter is used to associate the data source reference, in this case Dspostgres, to your caslib.

We should note here that you should not use the AddCaslibOptions.setDataSource methods. The SAS Viya 3.3 Java API for AddCasLibOptions does not have a utility method to associate relational data source instances with an AddCaslibOptions class. You must instead use the generic setParameter method with the AddCasLibOptions.KEY_DATASOURCE value to successfully assign a relational data source to an AddCasLibOptions class. If you do not add the data source instance to your AddCaslibOptions class, you will get the following error when the action is invoked:

```
     com.sas.cas.CASException: The action was not successful.
     (severity=2 reason=0 statusCode=2640504)
     5 ERROR: Missing path or data source options on caslib object pgLib.
     5 ERROR: The action stopped due to errors.
     debug=0x887fc1f8:PERM_MISSING_INFO
```

Now that we have the data source definition created and associated with an AddCaslibOptions action, it is time to create the caslib on the CAS server. We can create it by passing the AddCaslibOptions instance to the CASClient.invoke method. Here is an example:

```
     CASActionResults<CASValue> results = client.invoke(addCasLibOptions);
```

The results of invoking the addCasLibOptions are returned as a com.sas.cas.CASValue class. A simple toString() call on the CASValue object returns the following:

```
addCasLibOptions:
{
CASLibInfo=CASLibInfo
Name  Type     Description Path Definition         Subdirs Local Active Personal Hidden Transient
----- -------- ----------- ---- ------------------ ------- ----- ------ -------- ------ ---------
pgLib postgres                  uid = 'user'          0      1      1        0      0          0
pgLib                           pwd = '*****'         .      .      .        .      .          .
pgLib                           server = 'server'     .      .      .        .      .          .
pgLib                           schema = 'mySchema'   .      .      .        .      .          .
pgLib                           database = 'test'     .      .      .        .      .          .
5 rows
}
```

WORKING WITH CAS ACTIONS

A CAS action is a task that is performed by the CAS server at your request. The server parses the arguments of the request and invokes the action function. Actions that can be used with a data source caslib are loadTable, columnInfo, fileInfo, save, deleteSource, and fedsql.execdirect. In this paper, we will discuss loadTable, save, deleteSource, and fedsql.execdirect.

LOADTABLE ACTION

The loadTable action directs the server to load a table from a specified caslib data source into the CAS server. The action at a minimum takes a caslib parameter that describes the origin of the table to load, a path parameter that is the table name of the DBMS table to load, and a casout parameter for specifying the location for the output table. For example, if you have a table in your PostgreSQL DBMS named "cars" that you want to load into CAS with the name “mycars,” you could use the following PROC CAS loadTable statement:

```
     proc cas;
        session mysess;
        action loadTable / caslib="pgLib" path="cars" casout="mycars";
     quit;
```

In your Java program, you would use the com.sas.cas.actions.table.LoadTableOptions class to invoke this action. The above PROC CAS statement would map to the following Java code:

```
    LoadTableOptions loadTableAction = new LoadTableOptions();
    loadTableAction.setCaslib("pgLib"); // Set the input casLib
    loadTableAction.setPath("cars"); // identify that table to load
    loadTableAction.setCasOut(new Casouttablebasic().setName("mycars"));
    CASActionResults<CASValue> results = client.invoke(loadTableAction);
```

The set methods map one-to-one with the parameters of the loadTable action. The setCaslib and setPath are simple options, taking only a string as input; however, setCasOut is a complex option, taking a specific class, in this case com.sas.cas.actions.Casouttablebasic, which defines all the options for an output CAS table. The following code will pull the CASLogEvent from the CASActionResults class that is returned from invoking the loadTable action on your CAS client instance:

```
    List<CASLogEvent> l = results.getLogEvents();
    if (l.size() != 0) {
        Iterator<CASLogEvent> iter = l.iterator();
        while (iter.hasNext()) {
             CASLogEvent logEvent = iter.next();
             System.out.println(" " + logEvent);
        }
    }
```

The output log for the above example is as follows:

```
CASLogEvent [time=2018-02-20 18:17:07.125, type=3, message=NOTE: Performing
serial LoadTable action using SAS Data Connector to PostgreSQL.]
CASLogEvent [time=2018-02-20 18:17:07.686, type=3, message=NOTE: Cloud
Analytic Services made the external data from cars available as table mycars
in caslib pgLib.]
```

The output message is the same as you would see in your SAS log for PROC CAS.

Subset Your Columns

To subset the resulting columns from a loadTable action, you would pass in an array of com.sas.cas.actions.Casinvardesc instances to the loadTableOptions.setVars method. It is similar to setting the vars= option on the PROC CAS loadTable action. For example, if the "cars" table from the above example contained the columns "make," "model," "year," "color," and "units," but you were only interested in bringing the columns "color" and "make" into CAS, you would create the following Casinvardesc instances to pass to the loadTableOptions.setVars method:

```
    Casinvardesc storeID = new Casinvardesc().setName("color");
    Casinvardesc retailName = new Casinvardesc().setName("make");
    loadTableAction.setVars(new Casinvardesc[] {storeID,retailName});
```

Filter Your Data

The loadTableOptions.setWhere method enables you to filter data before it is loaded into a CAS table. The filtering occurs either in the database or on the CAS server as the data is read. If the target database is able to prepare the specified "where" statement, the filtering occurs in the database, and only filtered data is transferred to the CAS server. If the prepare call fails, the filtering of the data is performed on the CAS server. For example, Java code to subset on the "year" column and read in all the cars built in 2016 from the same "cars" table mentioned above is as follows:

```
    loadTableAction.setWhere("year = 2016");
```

Specify Data Source Options

The LoadTableOptions.setDataSourceOptions method allows you to set data source-specific options to apply during a loadTable action invocation. The setDataSourceOptions method takes a simple java.util.Map instance of key-value pairs. The keys are the data source option names, as documented in the SAS Viya System Programming Guide, and the values are the values to be applied to the specific data source option. For example, to set the number of read nodes to 3 for the above loadTable action, the Java code will be as follows:

```
Map<String, Object> datasourceOptions = new HashMap<String, Object>();
 datasourceOptions.put("numReadNodes", "3");
 loadTableAction.setDataSourceOptions(datasourceOptions);
```

SAVE TABLE ACTION

The Save table action allows you to write in-memory CAS tables back to your DMBS. To invoke this action, use the com.sas.cas.actions.table.SaveOptions class. At minimum, the action requires a caslib where the table is written, SaveOptions.setCaslib(), the name of the table to be created, SaveOptions.setName(), and a table object that describes the source table to write out, SaveOptions.setTable(). For example, if you wanted to write the “mycars” table that was loaded into CAS from the above loadTable action back to your PostgreSQL DMBS as “myCarsFromCAS”, the Java code would be as follows:

```
SaveOptions save = new SaveOptions();
save.setCaslib(“pgLib”);
save.setName("myCarsFromCAS");

Castable inTable = new Castable();
inTable.setCaslib("pgLib");
inTable.setName("mycars");

save.setTable(inTable);
CASActionResults<CASValue> results = client.invoke(save);
```

The PROC CAS call to perform the same Save action is as follows:

```
proc cas;
  session mysess;
   action save / caslib="pgLib" name="myCarsFromCAS"
       table={caslib="pgLib" name="mycars"};
run;
```

You can use the Castable.setWhere() and Castable.setVars() methods to filter data and columns that will be saved. For example, to subset the “cars” table to include only vehicle makes from 2016 and later, and create a new table in PostgreSQL named “my2016Cars” with only one column, named “makes,” issue the following two method calls on the Castable instance:

```
inTable.setWhere("year = 2016");
inTable.setVars(new Casinvardesc[] {new Casinvardesc().setName("make")});
```

DELETESOURCE ACTION

The deleteSource action, invoked on the com.sas.cas.actions.table.DeleteSourceOptions class, gives you the ability to drop a table from a caslib’s data source. In this example, you are dropping the “myCarsFromCAS” table that was created previously in the Save table section:

```
DeleteSourceOptions dropTable = new DeleteSourceOptions();
dropTable.setCaslib("pgLib");
dropTable.setSource("myCarsFromCAS");
CASActionResults<CASValue> results = client.invoke(dropTable);
```

The one slight difference between the DeleteSourceOptions class and the other action that we have discussed is that you call setSource() to specify the name of the table that you want to drop.

FEDSQL ACTION

The fedSQL action, invoked on the com.sas.cas.actions.fedsql.`ExecDirectOptions` class, allows you to load results of a SQL query against a data source caslib into a CAS table. To submit a fedSQL query to a data source caslib, make sure that the table referenced in the "from" clause is fully qualified using your caslib. For example, the following code will return all 2016 Fords with 6 cylinders from the "cars" table referenced by the PostgreSQL caslib "pgLib":

```
ExecDirectOptions fedSQL = new ExecDirectOptions();
fedSQL.setQuery("select make, model, Cylinders, msrp
                   from \"pgLib\".\"cars\"
                   where make='Ford' and Cylinders=6");
client.loadActionSet(fedSQL, fedSQL.getActionSetName());
CASActionResults<CASValue> results = client.invoke(fedSQL);
```

The results of the above invocation are as follows:

```
CASLogEvent [time=2018-02-25 14:41:53.138, type=3, message=NOTE: The
SQL statement was fully offloaded to the underlying data source via
full pass-through]

Key: Result Set
{
 Result Set=FedSQL Result Set
make model                          cylinders msrp
---- -------------------------- --------- -----
Ford  Explorer XLT V6                   6 29670
Ford  Escape XLS                        6 22515
Ford  Taurus LX 4dr                     6 20320
Ford  Taurus SES Duratec 4dr            6 22735
Ford  Freestar SE                       6 26930
Ford  Mustang 2dr (convertible)         6 18345
Ford  Taurus SE                         6 22290
7 rows
}
```

A message stating "NOTE: The SQL statement was fully offloaded to the underlying data source via full pass-through" indicates that the query was successfully passed down to the target database and executed. No additional SQL processing was done on the CAS server for the query.

CONCLUSION

SAS Viya provides many ways of interacting with other programming languages. With our relatively simple example, you now have the basic steps to leverage Java and SAS Viya to access your data sources. For more information, see "Getting Started with SAS® Viya™ for Java™"

REFERENCES

SAS Institute Inc. 2017. "Getting Started with SAS® Viya® for Java™." Available https://developer.sas.com/guides/java.html.

ACKNOWLEDGMENTS

The author extend his heartfelt thanks and gratitude to these individuals:

Joel Odom, SAS Institute Inc., Cary, NC

Jeff Bailey, SAS Institute Inc., Cary, NC

Susan Pearsall, SAS Institute Inc., Cary, NC

RECOMMENDED READING

Meng, Xiangxiang and Kevin Smith. 2017. “I Am Multilingual: A Comparison of the Python, Java, Lua, and REST Interfaces to SAS® Viya®.” *Proceedings of the SAS Global Forum 2017 Conference*. Cary, NC: SAS Institute Inc. Available http://support.sas.com/resources/papers/proceedings17/SAS0668-2017.pdf.

CONTACT INFORMATION

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

Salman Maher
100 SAS Campus Drive
Cary, NC 27513
SAS Institute Inc.
Salman.Maher@sas.com
https://www.sas.com/