Paper 4342-2016

High-Performance Data Access with FedSQL and DS2

Shaun Kaufmann, Farm Credit Canada

ABSTRACT

SAS® Federated Query Language (FedSQL) is a SAS proprietary implementation of the ANSI SQL:1999 core standard capable of providing a scalable, threaded, high-performance way to access relational data in multiple data sources. This paper explores the FedSQL language in a three-step approach. First, we introduce the key features of the language and discuss the value each feature provides. Next, we present the FEDSQL procedure (the Base SAS® implementation of the language) and compare it to PROC SQL in terms of both syntax and performance. Finally, we examine how FedSQL queries can be embedded in DS2 programs to merge the power of these two high-performance languages.

INTRODUCTION

In an enterprise environment it is common to have data spread across multiple relational database management systems (RDBMS). Additionally these datasets are often large and therefore the tables they reside in have been intentionally designed and implemented using techniques like indexing and partitioning to provide the best level of performance possible. Furthermore these databases usually run on very powerful hardware and simply put, it is in one's best interest to leverage these advantages to the fullest extent possible.

In this situation it is essential to have a shift in perspective regarding language selection and programming approach. Specifically, it is important to realize that we are no longer *working in SAS*. Instead we are *using SAS* to manipulate, join and subset remotely stored datasets. In fact the query results are often not intended to be stored in SAS at all but in one of these remote repositories. When combined with the fact that these dataset are often large in size, it becomes imperative to minimize the amount of data transferred back to the SAS server if we want to have code that runs efficiently. It is in this environment that the SAS Federated Query Language (FedSQL) and the DS2 procedure shine. Additionally, while it is possible to submit FedSQL programs in many ways, this paper focused on:

1. Submitting FedSQL programs from the Base SAS language interface by using the FEDSQL procedure.
2. Submitting FedSQL programs as part of a DS2 program.

These two options were chosen as they represent the opportunities for executing FedSQL programs from within a Base SAS session and therefore available to every SAS user.

Before moving on, it is pertinent to leave the reader with a few final comments regarding the intent of this paper. While both FedSQL and DS2 provide a wide array of features and function, this paper focuses on a subset specific to enable efficient data manipulation in the environment described above. Our focus is less about *how* to use these tools and more about *when* and *why* to use them. While we will start with a simple introduction to the basic syntax of each, the focus will be on the underlying concepts with resources for "learning the languages" identified along the way. Additionally, a discussion of these concepts often requires a background in areas such as DATA step, Proc SQL, SAS/ACCESS, and relational database management systems. While it is not feasible to provide that background as part of this paper, an attempt has been made to point the reader to the supporting documentation whenever possible in order to facilitate a more complete understanding of the material contained in this document.

FEDSQL – A MECHANISM FOR WRITING FEDERATED QUERIES

This paper explores the SAS FedSQL language as a mechanism to achieve efficient data access and manipulation in an environment where data is stored in *more than one* relational database management system. Specifically this paper focuses on an environment where:

- Data of interest is stored in datasets in multiple RDBMS.
- Those datasets are sufficiently large enough to make query execution duration of importance.
- There exists a desire to store the results of the data manipulations in a RDBMS while preserving the RDBMS specific data types (ANSI data types).

A SQL query that accesses data in this type of environment is referred to as a federated query. Here the federated query must be able to access data spread across multiple database systems without actual data integration, a concept known as data federation. In this situation the query engine must be capable of decomposing the query into sub queries for submission to the underlying databases and then appropriately recombining the result sets that are returned. This functionality is provided by the FedSQL language.

The four key benefits of the FedSQL language are listed in chapter three of the *SAS 9.4 FedSQL Language Reference*. Here we will quote these benefits verbatim before exploring what each really means.

Benefits of FedSQL

FedSQL provides many benefits if you are working in an environment in which you need more features than are provided in the SQL procedure.

- FedSQL conforms to the ANSI SQL:1999 core standard. This conformance allows it to process queries in its own language and the native languages of other data sources that conform to the standard.
- FedSQL supports many more data types than previous SAS SQL implementations. Traditional data source access through SAS/ACCESS translates target data source data types to and from two legacy SAS data types, which are SAS numeric and SAS character. When FedSQL connects to a data source, the language matches or translates the target data source's definition to these data types, as appropriate, which allows greater precision. Supported data types are described in "Data Types" on page 13.
- FedSQL handles federated queries. With the traditional DATA step or the SQL procedure, a SAS/ACCESS LIBNAME engine can access only the data of its intended data source.
- The FedSQL language can create data in any of the supported data sources, even if the target data source is not represented in a query. This enables you to store data in the data source that most closely meets the needs of your application.

BENEFITS OF FEDSQL (EXPLANATIONS AND EXPANSION)

Having read the benefits listed from chapter three of the FedSQL Language Reference many readers might be left thinking "That all sounds great, but what does it really mean to me"? This is a valid thought since the impact of each benefit can only truly be appreciated once a sufficient level of technical understanding is gained. To that end we explore each point in turn.

Point 1: FedSQL Conforms to the ANSI SQL:1999 Core Standard.

When a software vendor implements a structured query language within their product, they create functions and data types as part of the implementation. The details (names, parameters, return types, etc.) will be specific to their product…unless they adhere to an industry standard. ANSI SQL:1999 is one such standard. This ensures that there will exist a core set of functions and data types that will be supported by the RDBMS vendors that also adhere to this standard. This is important because having a common standard ensures that each sub query in a federated query can be pushed down to the RDBMS for processing. As we will see later this greatly reduces execution time.

Point 2: FedSQL Supports Many More Data Types than Previous SAS SQL Implementations.

What this really means is that FedSQL supports the ANSI data types mentioned above, which has many implications. First, this is necessary to ensure that the parameters and return types of ANSI functions can be accommodated. Second, this ensures a consistent level of precision for calculations, ensuring the "answer" that your FedSQL query produces is exactly the same as the source database would produce. Finally, having support for ANSI data types is imperative if you wish to create datasets within a RDBMS with the ANSI supported data types.

Point 3: FedSQL handles Federated Queries

As mentioned in the excerpt above, both the DATA step and the traditional SQL procedure rely on the SAS/ACCESS LIBNAME engine to provide their data access. And as mentioned, each SAS/ACCESS engine can only access data of its intended data source. This means that if you try to perform a create table query from a Select statement that references datasets from more than one vendor's RDBMS, then the SAS/ACCESS Engine must treat that query as three distinct operations: a create, a select, and an insert. This results in a large increase in the amount of data moving back to the SAS server and as a result a large increase in query duration. For more information on how the SAS/ACCESS engine works please see the *SAS/ACCESS 9.4 for Relational Databases Reference*. However the key point to remember is that FedSQL does NOT use the SAS/ACCESS engine to provide its connection to relational databases and therefore the restrictions mentioned in this paragraph DO NOT apply to FedSQL.

Point 4: The FedSQL language can create data in a data source that is not represented in the query.

This point seems obvious on the surface, however it has some subtleties that are not mentioned directly but that can be discovered through experimentation. What seems obvious is that even traditional DATA step and Proc SQL queries can create a dataset in a target database that was not part of the select query, so it seems odd to specifically call this out as a benefit of FedSQL. However experimentation shows that something is happening under the hood that greatly reduces the execution time required for a query of this nature. This suggests that FedSQL is likely designed in a way that allows federated query results to be transferred directly to the target database without passing through the SAS server first.

The Benefits of the FedSQL – Put into Practise

Having discussed what FedSQL provides in terms of benefits, it is now pertinent to explore how we can exploit these features in order to reduce query duration. At a high level our goal can be distilled into two overarching points. Specifically, in order to provide acceptable performance in a federated environment it is imperative that we:

1. Minimize the amount of data that gets transferred over your network.
2. Leverage the power of the RDBMS by taking advantage of its superior hardware and the fact that it was designed to efficiently manipulate its data (indexed, partitioned, etc.)

The reasoning for point one can be stated rather simply. Moving data over a network is slow when compared to moving data from disk to memory. Therefore every bit of data that is unnecessarily transferred over the network represents inefficiency and is to be avoided.

The reasoning for point two is less obvious without a sufficient understanding of relational database management systems. As mentioned in the introduction, RDBMS are specifically designed for high performance. They employ many advanced techniques that yield large performance benefits and they are typically installed on very powerful hardware. While the advanced techniques they employ is beyond the scope of this paper, a user is wise to remember that they do exist…and try to structure queries in a manner that takes advantage of the RDBMS' power.

FEDSQL FUNCTIONS – A BRIEF INTRODUCTION

As mentioned earlier, the point of this paper is *using SAS* to work with federated data, with an eye towards two key deliverables: speed of query execution, and storage of the resulting data sets within a RDBMS. This is achieved through the use of the FedSQL language and specifically the application of

ANSI functions to allow as much of the processing as possible to be pushed down to the RDBMS. Therefore it is important to understand what ANSI functions are available and what they do. While a comprehensive review of all the ANSI functions provided by the FedSQL language is beyond the scope of this paper, the duration of this paper will introduce several in order to demonstrate their power. For a complete review of all available functions the reader is directed to chapter five of the *SAS 9.4 FedSQL Language Reference document*.

THE FEDSQL PROCEDURE

The FEDSQL procedure enables you to submit FedSQL language statements from a Base SAS session and it will have a familiar feel to anyone experienced with the Proc SQL procedure. However there are also many differences that can trip a new user up. These include (but are not limited to) a different mechanism of data source connection, as well as differences in options, syntax and performance. The following sections explore these differences.

PROC FEDSQL VS PROC SQL – CONNECTIONS TO DATA SOURCES

Consider the following libname definition:

```
libname Oracle1 ORACLE PATH='DWP1' SCHEMA='IDS' USER='bob' PASSWORD='xxx';
```

Here we see a standard libname definition for the `ORACLE` SAS/Access engine. Here the `PATH` argument references my `DWP1` Oracle database and the `SCHEMA` argument references the `IDS` schema within the `DWP1` database. Similarly the `USER` and `PASSWORD` arguments specify the database user information required for database access. For the duration of this section we will assume the above libname statement has been executed and the connection is active.

Here it is important to be aware of what happens when a libname definition is executed. Specifically execution of the above libname starts a SAS/ACCESS interface that acts as an intermediary between a user's Base SAS code and the RDBMS. From this point forward, the SAS/ACCESS engine will examine any Base SAS code that the user executes against this data source and generates equivalent DBMS-specific SQL statements, passing the resulting SQL to the RDBMS for execution. To improve performance, the SAS/ACCESS engine breaks the query into query fragments and determines which fragments could be more efficiently processed by the RDBMS (as opposed to being processed by Base SAS). Any query fragments that cannot be passed to the RDBMS are processed in SAS.

What is important to take away from the above description is that the SAS/ACCESS engine controls the interactions between user code and the RDBMS. This is true regardless of whether the user code is written in the form of a DATA step or a Proc SQL procedure. However FEDSQL queries do not rely on SAS/ACCESS engines, but instead use a separate mechanism specifically designed for improved performance and compatibility when interacting with RDBMS data sources. As a consequence, anytime a Proc FEDSQL procedure executes it must first make its own connection to any required data sources. To enable ease of use the Proc FEDSQL procedure by default builds a connection string by leveraging all active librefs, since a FEDSQL connection requires the same information required to make a SAS/ACCESS connection. While it is possible (and occasionally desirable) to override this default behavior, leveraging a user's experience writing libname statements is the easiest way for them to get started with the Proc FEDSQL procedure. That said, the user needs to be aware that this mechanism only utilizes libref attributes that define for connection information RDBMS type and location. Any attributes that define behavior will not be used by the procedure. For additional information please see the *FedSQL Procedure* section of the *Base SAS 9.4 Procedure Guide*.

PROC FEDSQL VS PROC SQL - SYNTAX

This section provides a brief introduction to the FedSQL syntax with comparisons to the Proc SQL procedure. We examine examples introducing the concepts of ANSI data types and functions, as well as implicit and explicit pass-through queries. As is our practice throughout this paper, we will provide pointers to resources providing a more in-depth introduction to these topics for those readers without a strong background in SQL.

For the duration of this paper we shall assume the following two libref connections are active:

```
LIBNAME Oracle1 ORACLE PATH='DWP1' SCHEMA='IDS' USER='bob' PASSWORD='xxx';
LIBNAME TD1 TERADATA  SERVER="fccdrtst" DATABASE='DL_CSfB_Pop' USER='bob'
PASSWORD='xxx';
```

A Basic FedSQL Query

Writing a FedSQL procedure starts by encasing the desired query between the `Proc FedSQL` statement and the `QUIT` statement. Note that from a syntactic standpoint, a *simple* FedSQL query might not differ from a Proc SQL query. An example of a simple create table query is shown below for both a Proc SQL and Proc FedSQL implementation:

```
Proc FedSQL;
  Create Table TD1.table_B as select * from Oracle1.Table_A;
Quit;
```

```
Proc SQL;
  Create Table TD1.table_B as select * from Oracle1.Table_A;
Quit;
```

Here the similarity between the two procedures is evident. However, the reader should be aware that the syntax for an explicit pass-through is no longer identical, as illustrated in the following examples.

```
Proc FedSQL;
  Create Table TD1.table_B as select * from connection to Oracle1
    (Select * from Oracle1.Table_A);
Quit;
```

```
Proc SQL;
  Connect to Oracle (PATH='DWP1' SCHEMA='IDS' USER='bob' PASSWORD='xxx');
  Create Table TD1.table_B as select * from connection to oracle
    (Select * from Oracle1.Table_A);
  Disconnect from oracle;
Quit;
```

For readers unfamiliar with the concept of an explicit pass-though, here we are effectively telling the query engine exactly what to pass to the RDBMS. Additionally here we are free to (and required to) use the database's native SQL directly. For additional information regarding the explicit pass-through functionality, please see chapter three of the *SAS 9.4 FEDSQL Language Reference*.

PROC FEDSQL VS PROC SQL - PERFORMANCE

This section attempts to convey two main themes. First we endeavor to illustrate the type of situation a user might encounter while performing their day to day tasks. For example, consider a case where the user is writing a query to join two tables. Here it would not be beyond the realm of possibility to discover that the data types of the fields used in the join expression are inconsistent. When a problem of this nature occurs there can be many ways to solve it, which brings us to our second main theme: the performance impact of dealing with these situations in different ways. The duration of this subsection will explore the performance impact of dealing with this specific example in different ways.

Consider the following two table definitions, expressed as the DDL that would be used to create them.

```
Proc FedSQL;
  Create table TD1.Order_Table
  (
    Order_ID              Integer,
    Customer_ID           Integer,
    Order_Amount          Double
  );
Quit;
```

```
Proc FedSQL;
  Create table Oracle1.Customer_Table
  (
    Customer_Number       Varchar(10),
    Customer_Name         Varchar(60)
  );
Quit;
```

Here we find a situation where our table containing order information (called Order_Table) exists within aTeradata database and contains a field called Customer_ID of type Integer. Additionally we have a table containing customer data (called Customer_Table) that exists within an Oracle database. The customer table has a field called Customer_Number, which contains the same information as the Customer_ID field in Terdata. Unfortunately the data type of the Customer_Number field is varchar(10). Assuming that we wish to write a query to join the customer information with the order information we must somehow deal with these inconsistent data types. We now consider some alternatives and the implications that follow each.

Let us first consider a situation where we have the ability to modify the table structure of the customer tables. In this situation we could simply add an additional field (Customer _ID) to the table of the required data type (Integer). This can be easily handled through a few Proc FedSQL queries as follows:

```
/* Using the ALTER TABLE Statement to add a column of type Integer */
Proc FedSQL;
 Alter Table Oracle1.Customer_Table Add column Customer_ID Integer;
Quit;

/* Using the CAST Function to update a table. */
Proc FedSQL;
 Update Oracle1.Customer_Table Set Customer_ID = CAST(Customer_Number as Integer);
Quit;
```

Having modified the table structure using the `Alter Table` query and populating the new field the query with the ANSI CAST() function, writing a third query to join the two tables becomes trivial.

```
/* A Create Table query that joins the order and customer information */
Proc FedSQL;
 Create Table Oracle1.Orders_With_Customer_Info as
   Select A.Customer_Name, B.Customer_ID, B.Order_ID, B.Order_Amount
   From Oracle1.Customer_Table A, TD1.Order_Table B
   Where A.Customer_ID = B.Customer_ID;
Quit;
```

The above solution is quite reasonable, assuming that we have the authority to modify the Customer_Table in the Oracle Database. However, this is not a realistic expectation…especially if the Customer_Table resides in a production system. In this situation we need a more elegant solution that can accomplish the casting of the Customer_Number field to a type of Integer within the `Create Table` query itself. The following FedSQL does just that:

```
/* Employing the Cast() function as part of the join */
Proc FedSQL;
  Create table Oracle1.Orders_With_Customer_Info as
    Select A.Customer_Name, B.Customer_ID, B.Order_ID, B.Order_Amount
    From Oracle1.Customer_Table A, TD1.Order_Table B
    Where CAST(A.Customer_Number as Integer) = B.Customer_ID;
Quit;
```

The above FedSQL query achieves exactly the result that we were hoping for. However we could have accomplished the same result by using the INPUT() function in Proc SQL. So why would we want to use FedSQL instead? The answer is query performance. Consider the equivalent Proc SQL query presented below:

```
/* Employing the Input() function as part of the join in Proc SQL */
Proc SQL;
  Create Table Oracle1.Orders_With_Customer_Info as
    Select A.Customer_Name, B.Customer_ID, B.Order_ID, B.Order_Amount
    From Oracle1.Customer_Table A, TD1.Order_Table B
    Where INPUT(A.Customer_Number, 10.) = B.Customer_ID;
Quit;
```

Having run both queries we find that the FedSQL query executes in 1 minute and 15 seconds when the resulting dataset contained 1,500,000 rows. In comparison the Proc SQL query required 2 minutes and 2 seconds. This might not seem like much of a difference, however when the queries are rerun on larger datasets (where the output dataset contains 10,000,000 records) we find the run time required for the FedSQL query takes approximately 4 minutes while the Proc SQL query takes almost 10 minutes. Here it's clear to see that the difference becomes significant as data volume grows.

This performance difference is due to the difference in the way FedSQL handles fererated queries. Recall that by default SAS/ACCESS changes a `Create Table` query into separate `Create`, `Select` and `Insert` operations when it encounters a query containing more than one type of access engine. This results in all required data being transferred to the SAS server, followed by the resulting dataset being transferred to its intended location. However FedSQL appears to be able to avoid transferring data to the SAS server when all functions (like the CAST() function) can be processed directly by the database.

DS2 - A QUICK INTRODUCTION

The SAS DS2 language extends the traditional DATA step in much the same way that the FedSQL language extends the traditional SAS SQL offering. *The SAS 9.4 DS2 Language Reference* summarizes it as follows:

> DS2 is a new SAS proprietary programming language that is appropriate for advanced data manipulation. DS2 is included with Base SAS and intersects with the SAS DATA step. It also includes additional data types, ANSI SQL types, programming structure elements, and user-defined methods and packages. Several DS2 language elements accept embedded FedSQL syntax, and the runtime-generated queries can exchange data interactively between DS2 and any supported database. This allows SQL preprocessing of input tables, which effectively combines the power of the two languages.

While this paragraph accurately summarized the DS2 language in general, it fails to convey the magnitude of the benefits that can be achieved through the use of this new language. Comparable understatements could be achieved by saying that “a NASCAR extends the original Model-T” or that an F-22 Raptor extends a P-51 Mustang”. Each statement is technically correct, however the magnitude of the extension is not clear. So while a complete review of the DS2 procedure is beyond the scope of this paper it is helpful to be aware that the DS2 language is a significant advancement. The original DATA step was a “user” tool. The DS2 language is an “enterprise class” tool that allows SAS packages and applications to achieve the level of performance and functionality necessary to compete with other major players in the enterprise data and analytics space.

For readers without a background in DS2, it is useful to note that the basic purpose of the DS2 procedure is to provide a procedural programming environment similar in many ways to that of the DATA step. Specifically, both DS2 and the DATA step offer the user the ability to open a dataset and iterate through each row while applying programming logic to manipulate the data contained within it. This programming logic can take many forms including (but not limited to):

- program flow control structures like `IF` statements and `DO` loops
- array or hash object processing
- variable creation
- output control

Keeping this similarity in mind when exploring the DS2 language for the first time will allow the user to build on familiar concepts and greatly reduce the learning curve experienced. For a complete introduction to the SAS DS2 language please see the *SAS 9.4 DS2 Language Reference*.

FEDSQL AND DS2

This section focuses on the intersection between the FedSQL language and the DS2 language. Specifically, there are three options to utilize the FedSQL language in DS2 (not counting the use within hash objects). These options are: within a Set statement, within a SQLEXEC function, and within the SQLSTMT object. We will now explore each of these options and demonstrate an implementation of each.

THE SET STATEMENT

The *SAS 9.4 DS2 Language Reference* introduced the Set statement as follows:

> The SET statement is flexible and has a variety of uses in DS2 programming. These uses are determined by the options and statements that you use with the SET statement:
>
> - reading rows and columns from existing tables for further processing in a DS2 program
> - concatenating and interleaving tables, and performing one-to-one reading of tables

What the above statement really means is that anything you can do in a FedSQL query can now be achieved within the Set statement of a DS2 program. All the benefits discussed in the preceding sections of this paper are available in terms of performance and efficiency.

The following example illustrates a simple DS2 program that is functionally the same as the Proc FedSQL example discussed earlier. Here the FedSQL query in the Set statement joins a customer dataset stored in an Oracle database to an Orders dataset stored in a Teradata database. The FedSQL query again casts the Customer_Number field to an Integer to allow for the join operation to be performed successfully. Once the Set statement is executed, the Run statement processes each record in the recordset one at a time and in this case simply outputs them to the Orders_With_Customer_info_DS2 table that is created within the Oracle database associated with the Oracle1 libname.

```
Proc DS2;

  Data Oracle1.Orders_With_Customer_Info_DS2 (overwrite=yes);

    method run();

      set { Select A.Customer_Name, B.Customer_ID, B.Order_ID, B.Order_Amount
              From Oracle1.Customer_Table A, TD1.Order_Table B
             Where CAST(A.Customer_Number as Integer) = B.Customer_ID};

      output;

    end;

  enddata;

  run;
quit;
```

THE SQLEXEC FUNCTION AND SQLSTMT PACKAGE

The *SAS 9.4 DS2 Language Reference* introduces the SQLEXEC function and SQLSTMT package as follows:

The SQLSTMT package and the SQLEXEC function enable DS2 programs to dynamically generate, prepare, and execute FedSQL statements to update, insert, or delete rows from a table. With an instance of the SQLSTMT package or the SQLEXEXC function, the FedSQL statement allocate, prepare, execute, and free occurs at run time.

What this means is that through the use of these two components the full power of the FedSQL language is available at run time. That is, the programmer can dynamically create FedSQL queries on the fly throughout the execution of the DS2 program for whatever purpose is necessary. For example the programmer could create additional tables through the execution of DDL related statements, modify existing tables using an *alter table* query or dynamically constructed select or insert queries based on the specific values found in each record retrieved by the Set statement. The remainder of this section illustrates this power through the creation of a concrete example.

SQLEXEC FUNCTION

The SQLEXEC function is appropriate for executing a FedSQL statement that doesn’t return a recordset. Therefore it can be used to insert into, update or delete records from a table. Additionally, it can execute data definition statements that can create, modify and delete tables or views.

We will now consider an example that uses the SQLEXEC function to insert a record into a batch control table as part of a fictitious batch process. For the purposes of this example we will assume a batch control table already exists, having been previously created by the execution of the following FedSQL procedure.

```
Proc FedSQL;
  create table Oracle1.Batch_Table
  (
     Batch_ID              Integer,
     Time_Started          TIMESTAMP,
     Time_Ended            TIMESTAMP
  );
Quit;
```

From the FedSQL code above we can see that the batch control table is named Batch_Table and consists of 3 fields: a Batch_ID field of type integer, a Time_Started field of type Timestamp and a Time_Ended field of type TIMESTAMP. A full description of the TIMESTAMP data type can be found in the *SAS 9.4 FedSQL Language Reference*. However, for the purposes of this example we can simply say that the ANSI TIMESTAMP data type stores both date and time within a single element.

The following DS2 procedure extends our previous example by adding an init() method which automatically runs first when a DS2 procedure executes. Within the init() method we will perform three tasks. First, we declare a variable called `My_SQL_String` of type varchar. This variable will store the SQL statement to be executed. We then set the value of `My_SQL_String` variable to be a character string containing an insert statement that will insert the value 1 into the Batch_ID field, a timestamp value representing the current data and time into the Time_Started field and the value NULL into the Time_Ended field. Here it is important to note that `CURRENT_TIMESTAMP` actually is an ANSI datetime function, even though there aren't any parentheses () accompanying it. The `CURRENT_TIMESTAMP` function returned the current date and time in the form of a TIMESTAMP. Finally, we call the `SQLEXEC(My_SQL_String)`function to execute the FedSQL statement represented within the `My_SQL_String` variable. The complete code can be seen below.

```
Proc ds2;

  Data Oracle1.Orders_With_Customer_Info_DS2 (overwrite=yes);

    method init();
       Declare varchar(300) My_SQL_String;
       My_SQL_String = 'insert into Oracle1.batch_table (Batch_ID,
                        Time_Started, Time_Ended)
                        values (' || 1 || ', CURRENT_TIMESTAMP, NULL)';
      SQLEXEC(My_SQL_String);
    end;

    method run();
      set { Select A.Customer_Name, B.Customer_ID, B.Order_ID, B.Order_Amount
             From Oracle1.Customer_Table A, TD1.Order_Table B
             Where CAST(A.Customer_Number as Integer) = B.Customer_ID};

      output;
    end;

  enddata;
  run;
Quit;
```

SQLSTMT

The SQLSTMT package extends the functionality provided by the SQLEXEC function in that it is capable of returning a recordset that results from the execution of a FedSQL statement. We will now use the functionality of the SQLSTMT package to extend our prior example so that it is more useful. Previously, we had simply inserted a value of 1 into the Batch_ID field of the control table. However, our goal should actually be to determine the last (maximum) Batch_ID value and increment it by one. This can be accomplished as follows.

We must first declare a package variable of type SQLSTMT and instantiate it. Consider the following code snippet that declares a package variable `get_Last_Batch_ID` and instantiates it with the FedSQL statement `'Select max(Batch_ID) as Max_Batch_ID from Oracle1.batch_table'`.

```
Declare Package SQLSTMT get_Last_Batch_ID('Select max(Batch_ID) as
                                        Max_Batch_ID from Oracle1.batch_table');
```

Having declared and instantiated our SQLSTMT object, we can now execute it by calling its execute() method while storing its return code in a previously defined variable of type Integer called `rc_Execute`.

```
rc_Execute = get_Last_Batch_ID.execute();
```

If the value of `rc_Execute` is zero following the call to the execute() method, then the call was successful and we can retrieve the result set by calling the fetch() method to return the first row.

```
  rc_Fetch = get_Last_Batch_ID.fetch();
```

Again, if the return code of the `fetch()` method is zero we can be sure that a row was returned. We then call the `getInteger()` method to return the value of the first column of the result set into the Last_Batch_ID valuable, which was previously declared.

```
  get_Last_Batch_ID.getInteger(1, Last_Batch_ID, rc_getInteger);
```

Finally, we can update the code we previously wrote to construct the FedSQL statement contained in the `My_SQL_String` variable to use the value contained in the `Last_Batch_ID` variable.

```
My_SQL_String = 'insert into Oracle1.batch_table (Batch_ID, Time_Started,
                  Time_Ended ) values (' || Last_Batch_ID || ',
                  CURRENT_TIMESTAMP, NULL)';
```

The complete code now looks as follows, including all necessary variable definitions:

```
proc ds2;

  Data Oracle1.Orders_With_Customer_Info_DS2 (overwrite=yes);

     method init();

       /* Get the value last Batch_ID and increment it by one. */
       Declare Varchar(300) My_SQL_String;
       Declare Integer Last_Batch_ID rc_Execute rc_getInteger rc_getInteger;
       Declare Package SQLSTMT get_Last_Batch_ID('Select max(Batch_ID) as
                                       Max_Batch_ID from Oracle1.batch_table');

       Last_Batch_ID = 0;

       rc_Execute = get_Last_Batch_ID.execute();
       if (rc_Execute = 0) then do;
         rc_Fetch = get_Last_Batch_ID.fetch();
         if (rc_Fetch = 0) then do;
          get_Last_Batch_ID.getInteger(1, Last_Batch_ID, rc_getInteger);
         end;
       end;

       Last_Batch_ID = Last_Batch_ID + 1;

       /* Insert a record to indicate that we are starting our process. */

       My_SQL_String = 'insert into Oracle1.batch_table (Batch_ID,
                    Time_Started, Time_Ended ) values (' || Last_Batch_ID || ',
                    CURRENT_TIMESTAMP, NULL)';

       SQLEXEC(My_SQL_String);

      end;

      method run();

        set { Select A.Customer_Name, B.Customer_ID, B.Order_ID,
                         B.Order_Amount
               From Oracle1.Customer_Table A, TD1.Order_Table B
                Where CAST(A.Customer_Number as Integer) = B.Customer_ID
                and B.Customer_ID = 1};

        output;

      end;
     enddata;
    run;
    quit;
```

In this example we can see the power and flexibility of the two languages at work. Here DS2 is able to use a FedSQL query to return the maximum `Batch_ID` from a table residing in a remote database. Next it dynamically constructs a second FedSQL query to insert another record into that table with the incremented `Batch_ID` value. Finally the `run()`method uses a third FedSQL query to return a recordset from federated data sources and store the results directly back in the RDBMS.

CONCLUSION

Working with federated data can be a challenging task, especially when datasets are of significant size. In such an environment achieving satisfactory query performance requires both specialized tools and a non-trivial understanding of the underlying system architecture. However by focusing on using the FedSQL language intelligently, we are able to minimize the volume of data transferred over the network and harness the power of the relational database management systems. These two approaches to query development are of key importance when query performance is a primary goal. By employing the FedSQL language, and its implementation within the FedSQL and DS2 procedures, every Base SAS user has the power to achieve the query performance necessary to accomplish his or her goals.

REFERENCES

The SAS Institute. “Base SAS® 9.4 Procedures Guide, Fifth Edition”.

The SAS Institute. “SAS/ACCESS® 9.4 for Relational Databases Reference, Seventh Edition”.

The SAS Institute. “SAS® 9.4 DS2 Language Reference, Fourth Edition”.

The SAS Institute. “SAS® 9.4 FedSQL Language Reference, Third Edition”.

The SAS Institute. “SAS® 9.4 SQL Procedure User’s Guide, Third Edition”.

ACKNOWLEDGMENTS

I’d like to thank Christine Gamble and Suzanne Kaufmann for their efforts in editing this paper.

CONTACT INFORMATION

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

Shaun Kaufmann
Farm Credit Canada
Shaun.Kaufmann@gmail.com

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.