Paper 3734-2018

Anything You Can Do I Can Do Better: PROC FEDSQL VS PROC SQL

Cuyler R. Huffman[1,2], Matthew M. Lypka[1], Jessica L. Parker[1]

[1]Spectrum Health Office of Research, [2]Grand Valley State University

ABSTRACT

Structured Query Language (SQL) was implemented in SAS as PROC SQL. A benefit of the SQL Procedure is that you can write queries or execute SQL statements on a SAS dataset or in a database. Another benefit is the SQL language makes it capable to combine the functionality of a DATA step and multiple PROC steps all into one procedure. Although useful, PROC SQL is limited in that it can only make one database connection per query, and is not compliant with the American National Standards Institute (ANSI) SQL syntax. Due to this non-compliance interacting with ANSI standard compliant databases becomes more difficult. Due to these limitations with PROC SQL a new procedure, PROC FEDSQL, was introduced.

PROC FEDSQL was introduced in SAS 9.4, and offers faster performance, ability to connect to multiple databases in one query, increased security, broader support of data types, and full compliance with the ANSI SQL: 1999 core standard, among others. In this paper, we will explore these acclaimed benefits. We will compare PROC SQL and PROC FEDSQL in terms of syntax, performance, and output. We will determine which procedure should be used in various use cases and how and when to incorporate PROC FEDSQL into your workflows.

INTRODUCTION

Introduction of PROC SQL

Structured Query Language (SQL) was first developed in the early 1970’s. The purpose of SQL was to manage data stored in Relational Database Management Systems (RDBMS). In 1986 SQL became the standard language of the American National Standards Institute (ANSI) and this standard SQL language has been revised and updated over the years. SQL was introduced into SAS as PROC SQL.

The major benefit to using PROC SQL over a DATA step or other procedure steps is that PROC SQL could combine the functionality of the DATA and PROC steps into a single step. This often leads to fewer lines of code than the traditional DATA and procedure steps coding style. One of the other benefits of PROC SQL is the ability to connect to, retrieve, update, and report information from other RDBMS. In order to connect to RDBMS some of the database features associated with SQL were not implemented with PROC SQL, therefore, PROC SQL is not fully compliant with ANSI standards.

Introduction of PROC FEDSQL

A Federated Query Language (FEDSQL) can simultaneously connect to multiple databases all under a single query request. However, in order for this connection, and associated query, to happen all of the databases must be coordinated under the same standard. Therefore, when SAS introduced PROC FEDSQL in SAS 9.4 they made PROC FEDSQL to fully conform to ANSI SQL 1999 standards and allows the processing of queries in the native languages of all other data sources that also conform to the ANSI 1999 standard. Previously users using PROC SQL were limited to using only character, integer, decimal, and date data types. With PROC FEDSQL and the compliance now the user has access to many more data types including, but not limited to:

- Time(p)
- Timestamp(p)
- Double
- NChar
- Varchar

With the support of the new data types comes the ability to create datasets within a RDMNS, which is also compliant with ANSI 1999 standards, using PROC FEDSQL (The SAS Institute, 2018).

More benefits of PROC FEDSQL

Along with the ANSI compliance and all the associated benefits that come along with it, PROC FEDSQL brought many other benefits to SAS users. PROC FEDSQL enables the user to access multiple data sources all within one query, as opposed to a different query for each new data source that PROC SQL would do previously. Multiple data sources are combined in a query in PROC FEDSQL when one or more SELECT statements are used to produce a dataset. If multiple SELECT statements are used in a query then their resulting sets must be combined in some way using set operators so the query produces a single output set as a result. Another benefit that comes with PROC FEDSQL is a SQL pass-through process where the query response time is reduced and the security is enhanced. There are two types of pass-through that PROC FEDSQL can perform, explicit and implicit pass-through. Explicit pass-through has the user connect to a data source and then send that data source SQL statements directly for that data source to execute. An associated benefit of this is that the user is able to use the syntax that is native to that data source even if that syntax is non-ANSI standard. An implicit pass-through, on the other hand, takes SAS syntax and translates it into equivalent code specific to the data source the user is connected to, therefore, the data can be passed to the data source directly for processing. Due to the fact that the data source is processing the query, the required data does not need to be transferred to a SAS server, only the resulting dataset needs to be transferred, which greatly reduces query time (The SAS institute, 2018). Since the query is being processed within the data source and only the resulting table is being transferred back, this eliminates the need of having to transfer over tables that may contain sensitive information. The SQL Procedure also has this capability, however, by default, the FEDSQL Procedure attempts to use an implicit-pass through for all SQL data sources.

DIFFERENCES BETWEEN PROC SQL AND FEDSQL

Many SAS users may not be familiar with PROC FEDSQL, but with the promise of better performance one might expect that they can go to any of their previously written PROC SQL code and then add “FED” to the top and rerun it for instant benefits. The example code below is straight from the Base SAS 9.2 Procedures Guide. All the code is doing is creating a table in Proc SQL and then inserting data into it.

```
Proc sql;
   create table proclib.paylist
       (IdNum char(4),
        Gender char(1),
        Jobcode char(3),
        Salary num,
        Birth num informat=date7.
                  format=date7.,
        Hired num informat=date7.
                  format=date7.);
insert into proclib.paylist
    values('1639','F','TA1',42260,'26JUN70'd,'28JAN91'd)
    values('1065','M','ME3',38090,'26JAN54'd,'07JAN92'd)
    values('1400','M','ME1',29769.'05NOV67'd,'16OCT90'd)
    values('1561','M',null,36514,'30NOV63'd,'07OCT87'd)
    values('1221','F','FA3',.,'22SEP63'd,'04OCT94'd);
quit;
```

Figure 1

Now we copied the previous PROC SQL code and added “FED” to the beginning so we can see if it does exactly what PROC SQL did just better.

```
Proc fedsql;
   create table proclib.paylist
       (IdNum char(4),
        Gender char(1),
        Jobcode char(3),
        Salary num,
        Birth num informat=date7.
                  format=date7.,
        Hired num informat=date7.
                  format=date7.);
insert into proclib.paylist
    values('1639','F','TA1',42260,'26JUN70'd,'28JAN91'd)
    values('1065','M','ME3',38090,'26JAN54'd,'07JAN92'd)
    values('1400','M','ME1',29769.'05NOV67'd,'16OCT90'd)
    values('1561','M',null,36514,'30NOV63'd,'07OCT87'd)
    values('1221','F','FA3',.,'22SEP63'd,'04OCT94'd);
quit
```

Figure 2

```
ERROR: Syntax error at or near "informat"
84   insert into proclib.paylist
85       values('1639','F','TA1',42260,'26JUN70'd,'28JAN91'd)
86       values('1065','M','ME3',38090,'26JAN54'd,'07JAN92'd)
87       values('1400','M','ME1',29769.'05NOV67'd,'16OCT90'd)
88       values('1561','M',null,36514,'30NOV63'd,'07OCT87'd)
89       values('1221','F','FA3',.,'22SEP63'd,'04OCT94'd);
ERROR: Syntax error at or near "D"
90   select *
91      from proclib.paylist;
ERROR: Table "PROCLIB.PAYLIST" does not exist or cannot be
accessed
ERROR: BASE driver, Table PAYLIST does not exist or cannot be
accessed or
       created
```

Figure 3

Unfortunately, that doesn't seem to be the case. This begs the question, "If I can't just replace PROC SQL with PROC FEDSQL how much work am I going to have to put into this in order to see those benefits?" To answer this question, we will look at everyday scenarios that an average SAS programmer may face and see just how different PROC SQL and PROC FEDSQL really are. Our criteria for this are Syntax, Performance, and Output.

Syntax: Supported Data Types

People that have knowledge of PROC SQL will find that much of the syntax of PROC FEDSQL is familiar. However, there are a few intricacies between the two procedures that should be highlighted. The first of which is the data types that are supported within PROC FEDSQL. For instance, we can see from the example code in Figure 1 that PROC FEDSQL can create data types like double, as well as others that are compliant with the ANSI standard. The inclusion of more data types allows PROC FEDSQL more precision. For instance, if querying data that is of the data type BIGINT, which is supported in PROC FEDSQL, in PROC SQL the corresponding results may be inaccurate. The reason for this is due to the fact that PROC SQL does not have the precision to process integers larger than a certain size, therefore, it will display it as floating point notation, not be able to display it at all, or not display it correctly (Mohammed, Gangarajula, Kalakota, 2015). This can ultimately influence results and output if the issue is not caught. It is important to know the data types of the data that is being queried when connecting to outside databases.

```
*Proc FEDSQL;
create table sales (prodid double not null,
                             custid double not null,
                             totals double having format comma8.,
                             country char(30));


*PROC SQL;
create table sales2 (prodid numeric,
                                      custid numeric,
                                      totals numeric format
comma8.,
                                      country char(30));
```

Figure 4

Syntax: Making Connections

One of the main benefits of both PROC FEDSQL and PROC SQL is the ability to connect to other RDBMSs, for instance Oracle or MySQL. The syntax changes depending on whether an implicit or explicit pass through is being performed. One of the main differences between the two is that an implicit pass through uses a LIBNAME statement in order to connect to the other data sources. After using the LIBNAME statement the connection will be made whenever libref name is used. In the example a LIBNAME is being used to connect to the ORACLE engine, notice that all the connection information needed to access the Oracle database needs to be specified. Syntax wise when PROC SQL and PROC FEDSQL make a connection this way it is nearly identical.

```
*Implicit Pass-through;
libname myspde spde 'C:\spde';
libname myoracle oracle path=ora11g user=xxxxxx password=xxxxxx
schema=xxxxxx;
proc fedsql;
   select * from myspde.product
      where exists (select * from myoracle.sales
      where product.prodid=sales.prodid);
quit;
```

Figure 5

An explicit pass-through, however, does not use a LIBNAME statement. Instead an explicit pass-through makes these connections within the procedure. This is done using the CONNECT TO statement where all the connection information needed to access the database needs to be specified. Once the connect to statement is specified, the corresponding query within the statement is being passed directly to that data source for execution. Another important aspect to explicit pass through is the EXECUTE statement. The EXECUTE statement passes the SQL statements that are specific to the RDBMS that was specified in the CONNECT TO statement for processing. Statements included in EXECUTE statements are specifically non-query statements. Instead, the purpose is to manipulate the table being accessed in some way, for instance deleting/inserting rows, updating data, etc. Similar to implicit pass-through, the syntax to perform explicit pass through is very similar between the two procedures. However, as mentioned previously, PROC FEDSQL has the added capability of being able to perform federated queries and in those instances the syntax changes between PROC FEDSQL and PROC SQL. We can see that PROC FEDSQL is able to perform this query all within one select statement while querying data from both the Microsoft SQL Server and Oracle. For PROC SQL to perform this same query, multiple select statements and a join would have to be performed.

```
*Explicit Pass-Through;
proc sql;
   connect to oracle as ora2 (user=user-id password=password);
   select * from connection to ora2 (select lname, fname, state from staff);
   disconnect from ora2;
quit;

*Execute statement;
proc sql;
   connect to oracle(user=user-id password= password);
   execute (create view whotookorders as select ordernum, takenby,
     firstname, lastname,phone from orders, employees
     where orders.takenby=employees.empid) by oracle;
   execute (grant select on whotookorders to testuser) by oracle;
disconnect from oracle;
quit;
```

Figure 6

```
*Federated Query;
LIBNAME MSSQL ODBC DSN="MSSQLSERVER";
libname myoracle oracle path=ora11g user=xxxxxx password=xxxxxx
schema=xxxxxx;
proc fedsql;
 create table payment as select S.ID,
                                O.Transaction,
                                S.Amount,
                                O.Product
                from mssql.product S, myoracle.sales O
      where S.prodid= O.prodid);
quit;

*Federated Query;
LIBNAME MSSQL ODBC DSN="MSSQLSERVER";
libname myoracle oracle path=ora11g user=xxxxxx password=xxxxxx
schema=xxxxxx;
Proc SQL;
create table payment1 as select mssql.ID,
               mssql.Amount,
               mssql.prodid
               from  mssql.product;
create table payment2 as Select myoracle.Transaction,

               myoracle.Product,
                                myoracle.PRODID
                                      from myoracle.sales;
Create table final as select * from
payment2 as d1 full join
payment1 as d2 where d1.prodid = d2.prodid;
quit;
```

Figure 7

Output: Typical Two-Table Join

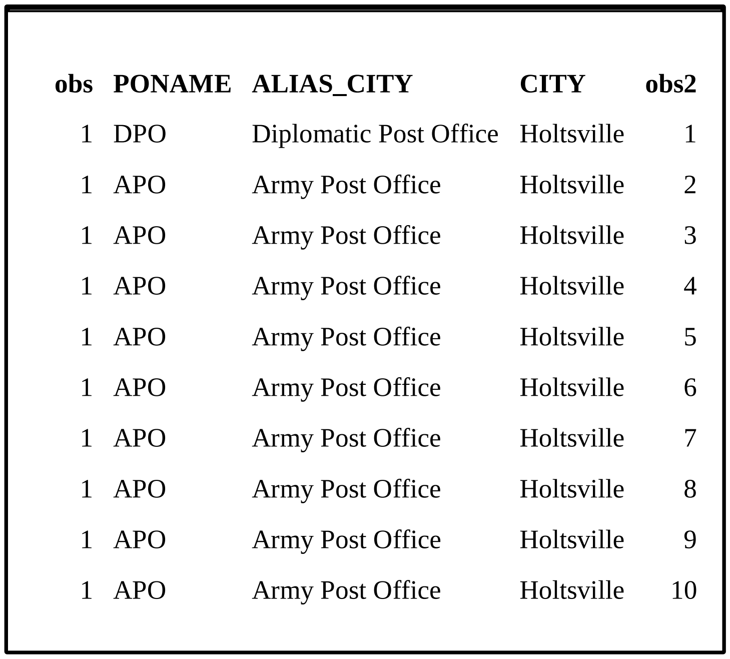
For this example, we are using two SASHELP datasets, Zipcode and Zipmil, and joining them together. This new dataset is not practical when it comes to everyday use, but it does highlight something interesting going on in the background when running PROC FEDSQL and PROC SQL. To make it clear as to what exactly is happening, an observation number was added in both the Zipcode and Zipname datasets; these are called OBS and OBS2, respectively. Outputting the first ten observations from each of the queries we notice that PROC FEDSQL and PROC SQL perform the same join differently. The resulting tables for the PROC FEDSQL and PROCSQL queries are represented in Figure 9 and Figure 10 respectively.

```
*Equijoin;
proc fedsql;
create table zip3 as select zipcode.obs, ZIPMIL.PONAME,ZIPMIL.ALIAS_CITY,
ZIPCODE.CITY, zipmil.obs2
from zipcode, zipmil
where zipcode.STATENAME = zipmil.STATENAME;
quit;

*Equijoin;
proc sql;
create table zip4 as select zipcode.obs, ZIPMIL.PONAME,ZIPMIL.ALIAS_CITY,
ZIPCODE.CITY, zipmil.obs2
from zipcode, zipmil
where zipcode.STATENAME = zipmil.STATENAME;
quit;
```

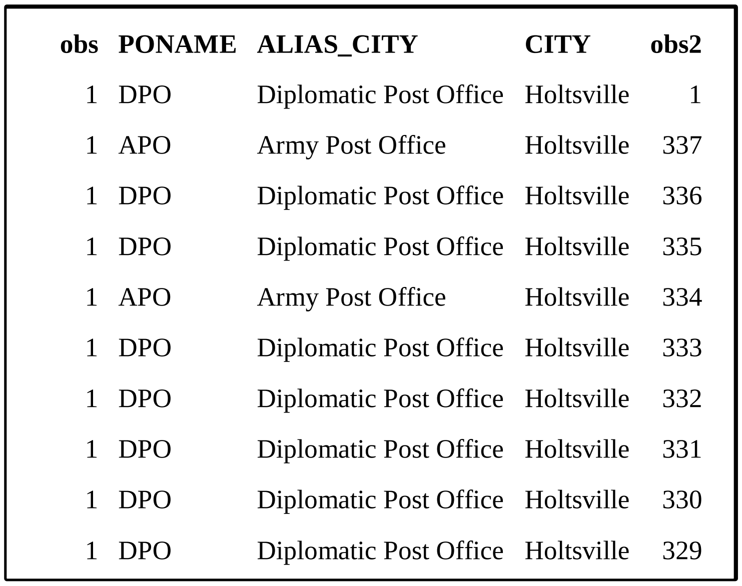
Figure 8

We can see from Figure 9 that PROC FEDSQL performs this join exactly as one might expect. All of the columns specified from the first dataset, in this case the Zipname dataset with the specified columns PONAME, Alias_City, and OBS2, are combined with the first row of these selected columns with the first row from the second table or dataset, in this case the Zipcode dataset with the variables City and OBS. Then the same columns from the Zipname dataset are then joined with the second row of the Zipcode data set, and so on and so forth.



obs	PONAME	ALIAS_CITY	CITY	obs2
1	DPO	Diplomatic Post Office	Holtsville	1
1	APO	Army Post Office	Holtsville	2
1	APO	Army Post Office	Holtsville	3
1	APO	Army Post Office	Holtsville	4
1	APO	Army Post Office	Holtsville	5
1	APO	Army Post Office	Holtsville	6
1	APO	Army Post Office	Holtsville	7
1	APO	Army Post Office	Holtsville	8
1	APO	Army Post Office	Holtsville	9
1	APO	Army Post Office	Holtsville	10

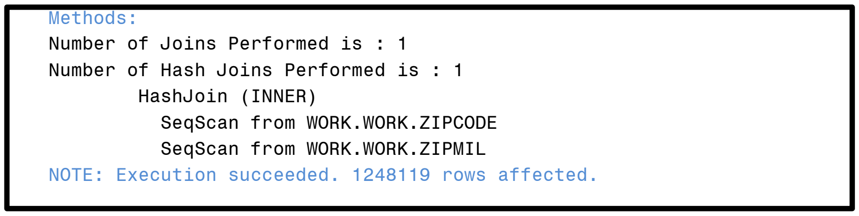
Figure 9



obs	PONAME	ALIAS_CITY	CITY	obs2
1	DPO	Diplomatic Post Office	Holtsville	1
1	APO	Army Post Office	Holtsville	337
1	DPO	Diplomatic Post Office	Holtsville	336
1	DPO	Diplomatic Post Office	Holtsville	335
1	APO	Army Post Office	Holtsville	334
1	DPO	Diplomatic Post Office	Holtsville	333
1	DPO	Diplomatic Post Office	Holtsville	332
1	DPO	Diplomatic Post Office	Holtsville	331
1	DPO	Diplomatic Post Office	Holtsville	330
1	DPO	Diplomatic Post Office	Holtsville	329

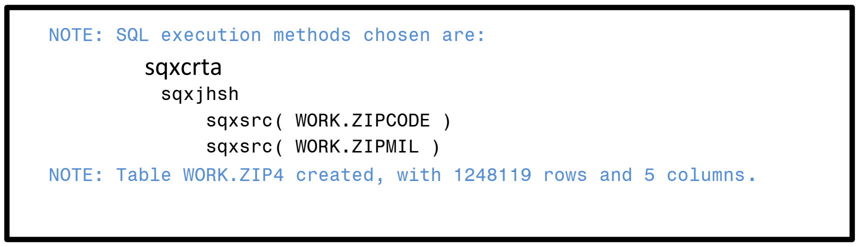
Figure 10

However, what is interesting is that PROC SQL first joins the first row as expected, DPO and Diplomatic Post Office are joined with the first row of the Zipcode dataset where the City is Holtsville. Then, instead of moving to the second row of the Zipmil dataset, PROC SQL moves to the last row of the Zipmil dataset where PONAME and Alias_city are APO and Army Post office and are joined onto the first row of the Zipcode dataset, where City is Holtsville. PROC SQL continues in this backwards fashion until it has gone through every row of the Zipname dataset, then moves on to row two of the Zipcode dataset, again starting with row one, then the last row, then the backwards process from there. Using the _Method option, the methods both procedures used to perform this join are outputted to the log. It is easy to see that PROC FEDSQL is performing a hashjoin between the two datasets. However, with PROC SQL, interpreting what methods were used is more difficult. Notice the SQXCRTA and the SQXJHSH in the PROC SQL log method details, what these methods are doing is creating a table and then performing a hash join, respectively. Therefore, up until this point, both PROC FEDSQL and PROC SQL are doing things the same. The differences come in to play when going through the rows of each of the tables. The method that PROC SQL uses to do this is SQXSRC which will source rows from the two tables. The FEDSQL Procedure, on the other hand, is using the SeqScan Method. The SeqScan Method is reading in the data in sequential order. It is due to this sequential ordering that there are differences in the output. PROC SQL is making this join as efficiently as possible, which is why it goes to the first row, then the last row and so on, while PROC FEDSQL just does it in order.



```
Methods:
Number of Joins Performed is : 1
Number of Hash Joins Performed is : 1
        HashJoin (INNER)
          SeqScan from WORK.WORK.ZIPCODE
          SeqScan from WORK.WORK.ZIPMIL
NOTE: Execution succeeded. 1248119 rows affected.
```

Figure 11 FEDSQL Methods

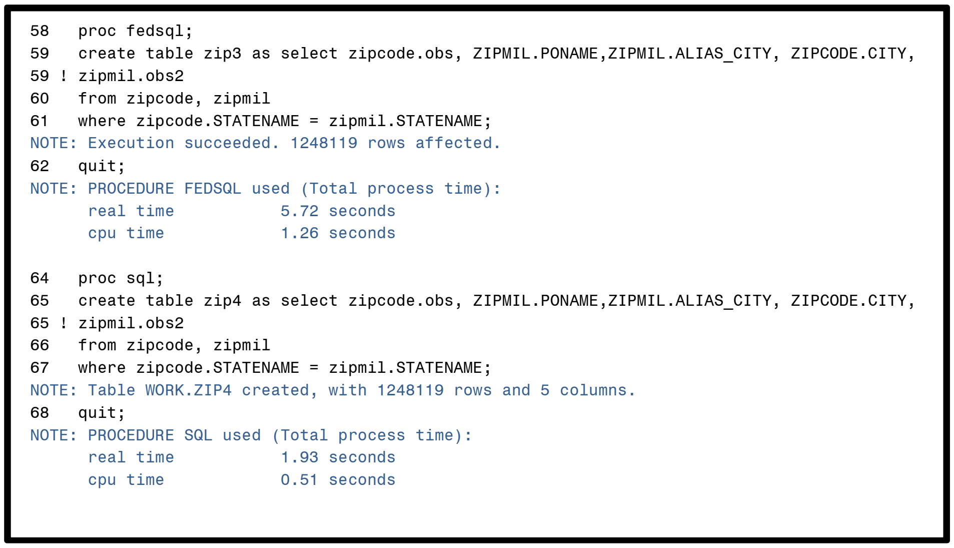


```
NOTE: SQL execution methods chosen are:
        sqxcrta
          sqxjhsh
              sqxsrc( WORK.ZIPCODE )
              sqxsrc( WORK.ZIPMIL )
NOTE: Table WORK.ZIP4 created, with 1248119 rows and 5 columns.
```

Figure 12 SQL Methods

Performance

One of the associated benefits of PROC FEDSQL is that the query times are reduced. This is done when connecting to other databases and using the implicit pass through process to its fullest potential. However, the query time reduction does not seem to hold up when creating and running the query all within a local SAS environment. One of the reasons for this is the sequential scan that PROC FEDSQL is performing when joining tables. We know that ordering can be very resource intensive, therefore, PROC SQL is performing joins as efficiently as possible when no order by statement is specified. We can see from Figure 13 that the real time for PROC FEDSQL to run the query was 5.72 seconds while PROC SQL was able to perform this same query in 1.93 seconds, again in real time. While four seconds in real time may not have much of a significant impact we can see that if we are working with large datasets that these time differences would become noticeable.



```
58   proc fedsql;
59   create table zip3 as select zipcode.obs, ZIPMIL.PONAME,ZIPMIL.ALIAS_CITY, ZIPCODE.CITY,
59 ! zipmil.obs2
60   from zipcode, zipmil
61   where zipcode.STATENAME = zipmil.STATENAME;
NOTE: Execution succeeded. 1248119 rows affected.
62   quit;
NOTE: PROCEDURE FEDSQL used (Total process time):
      real time           5.72 seconds
      cpu time            1.26 seconds

64   proc sql;
65   create table zip4 as select zipcode.obs, ZIPMIL.PONAME,ZIPMIL.ALIAS_CITY, ZIPCODE.CITY,
65 ! zipmil.obs2
66   from zipcode, zipmil
67   where zipcode.STATENAME = zipmil.STATENAME;
NOTE: Table WORK.ZIP4 created, with 1248119 rows and 5 columns.
68   quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           1.93 seconds
      cpu time            0.51 seconds
```

Figure 13

We can start seeing a drastic change in performance when we move into correlated subqueries. For those unfamiliar, a correlated subquery is a query nested inside another query, and the nested query is using values from the outer query. In SAS, this is done with a WHERE clause. The WHERE clause within the subquery is referring to values from the outside query.

```
proc sql  _method;
create table sub4 as
select *
from zipcode e
where exists(select * from zipmil d
             where d.statename = e.statename);
NOTE: SQL execution methods chosen are:
    sqxcrta
          sqxfil
              sqxsrc( WORK.ZIPCODE(alias = E) )
NOTE: SQL subquery execution methods chosen are:
          sqxsubq
              sqxsrc( WORK.ZIPMIL(alias = D) )
NOTE: Table WORK.SUB4 created, with 6217 rows and 22 columns.
quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.14 seconds
      cpu time            0.04 seconds
```

Figure 14

```
proc fedsql _method;
NOTE: Writing HTML Body file: sashtml.htm
create table sub3 as select * from zipcode e
where exists(select * from zipmil d
             where d.statename = e.statename);
Methods:
        SeqScan with qual from WORK.WORK.ZIPCODE
          SubPlan (EXISTS) in qual
            SeqScan with qual from WORK.WORK.ZIPMIL
NOTE: Execution succeeded. 6217 rows affected.
quit;
NOTE: PROCEDURE FEDSQL used (Total process time):
      real time           26.89 seconds
      cpu time            19.01 seconds
```

Figure 15

As we can see from Figure 14 and Figure 15, the real time taken to run our correlated subquery in PROC FEDSQL was 26.89 seconds, the amount of time it took PROC SQL to run the same query was 0.14 seconds. Using the _METHOD statement again we can see that the reason for this difference in time is due to the fact that PROC FEDSQL and PROC SQL are using different methods to perform this same query. In the previous example it was highlighted that this SeqScan method is increasing query time, however, now there is a new method SubPlan that is bogging down the speed. For many SQL systems a Subplan is part of maintenance plan and work similarly to organizing tools. PROC SQL on the other hand uses the SQXFIL and SQXSUBQ methods which stand for row filtration and subquery respectively. However, it is important to note that PROC FEDSQL has the ability to perform subqueries from two different data sources. It does this by directing the subquery to the data source to be performed there, this reduces performance time because the data source is doing all the work and only the resulting table or data set is being transferred from the data source into SAS. While PROC SQL is able to perform correlated subqueries, it is unable to perform them from multiple data sources.

CONCLUSION

Breaking the comparison between PROC FEDSQL and PROC SQL down into syntax, performance, and output when evaluating which to use in everyday use cases, we can see that PROC FEDSQL does not do “everyday” tasks nearly as well. However, that is justifiable; FEDSQL is excellent at doing what it was created to do which is connecting multiple different databases at once. The benefits associated with PROC FEDSQL in the times that PROC FEDSQL is needed far outweigh anything that PROC SQL has to offer. However, those looking to replace PROC SQL with PROC FEDSQL are going to be disappointed because PROC SQL is simply better when it comes to everyday use cases, like creating tables, joining tables, performing subqueries, when considering performance time. However, when working with other RDBMSs that are outside of SAS, this is when PROC FEDSQL really shines. Even though PROC FEDSQL and PROC SQL look similar, and even sound similar, it is important to keep in mind that they were created to perform different tasks. PROC FEDSQL was not created to replace PROC SQL, it was created to perform tasks that PROC SQL, for one reason or another, was not able to perform or not able to perform well. Therefore, we see that PROC FEDSQL cannot do everything PROC SQL can do better, because it wasn’t meant to.

REFERENCES

Dickstein, Craig, & Pass, Ray (2003) "DATA Step vs. PROC SQL: What's a neophyte to do?". Available at <http://www2.sas.com/proceedings/sugi29/269-29.pdf>

Kaufmann, Shaun (2016) "High-Performance Data Access with FedSQL and DS2". Available at <https://support.sas.com/resources/papers/proceedings16/4342-2016.pdf>

Lafler, Kirk (2009) "Exploring the Undocumented PROC SQL _METHOD Option". Available at <http://support.sas.com/resources/papers/proceedings09/063-2009.pdf>

Mohammed, Z., Gangarajula, G., Kalakota, P. (2015) "Working with PROC FEDSQL in SAS® 9.4". Available at <https://support.sas.com/resources/papers/proceedings15/3390-2015.pdf>

Ronk, Katie (2003) "Introduction to Proc SQL". Available at <http://www2.sas.com/proceedings/sugi29/268-29.pdf>

The SAS Institute. "Base SAS® 9.2 Procedures Guide".

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

The SAS Institute. "SAS/ACCESS® 9.4 for Relational Databases: Reference, Ninth Edition"

The SAS Institute. "Base SAS® 9.4 Procedures Guide, Seventh Edition"

The SAS Institute. "SAS® 9.3 SQL Procedure User's Guide"

The SAS Institute. "SAS/ACCESS® 9.3 for Relational Databases: Reference, Second Edition"

CONTACT INFORMATION

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

Cuyler Huffman
Spectrum Health Office of Research
Cuyler.Huffman@SpectrumHealth.org

Matthew Lypka
Spectrum Health Office of Research
Matthew.Lypka@SpectrumHealth.org

Jessica Parker
Spectrum Health Office of Research
Jessica.Parker2@spectrumhealth.org