PharmaSUG 2019 - Paper AP-290

PROC SQL to the Rescue: When a Data Step Just Won't Do Anymore

John R. O'Leary, Yale University School of Medicine

ABSTRACT

This paper is written for beginning and intermediate Base SAS® users who are more comfortable using the traditional data step but would benefit from efficiencies gained by PROC SQL, particularly when it is used explicitly. When processing datasets with 100 million records or more, there reaches a point when a programmer is forced to try new tools if they want to get home in time for supper. This paper provides some PROC SQL examples and other time-saving programming tips with emphasis on the SQL Procedure Pass-Through Facility. Although PROC SQL can be run implicitly within the SAS® Application Server, it also can interface directly with various relational database management systems (RDBMS) such as ORACLE, DB2, MySQL and Microsoft® SQL Server so that the SQL code runs directly on the RDBMS, thereby reducing the amount of data that needs to be transferred back to SAS®. The examples for this paper are limited to the syntax applicable to Microsoft® SQL Server and require the installation of SAS/ACCESS® Interface to Microsoft® OLEDB.

INTRODUCTION

When do you realize that you need to expand your skills and employ new methods of querying your data? How about when you can no longer run your queries during the day but instead choose to let them run overnight because they are taking more than 10 hours to complete? So, just like the day when it became apparent that PROC REPORT offered benefit over my tried-and-true reporting tool PROC TABULATE, it finally became clear that the trusty data step was not the correct instrument for all tasks. As a SAS® programmer who has typically worked with data sets for smaller medical research studies (1000 patients or less), the data step had worked wonderfully for these past projects. However, after being requested to obtain the vital sign data from a SQL Server electronic medical record system for millions of Veterans so that it could be analyzed in SAS®, the sheer volume of data exceeded the limits of using a typical data step or even trying to run standard implicit PROC SQL code. Fortunately, colleagues suggested using the PROC SQL Pass-Through Facility, noting it was likely to reduce the program run time significantly. But wait this would mean having to delve into SQL syntax native to Microsoft® SQL Server. Given that SQL is a standard programming language that follows standards across database products, figuring out the nuances of the SQL for your RDBMS (in this case SQL Server) is not that difficult given all the online resources.

QUICK WORD ABOUT ARCHITECTURE

While understanding the hardware and network architecture may be of little interest to many programmers, there are a couple of key points that are worth considering. First, moving large amounts of data across the network between the SAS® server and the RDBMS will definitely slow up your query processing. When possible, it is a good idea to reduce the amount of data traffic and perform the primary processing on the source database system particularly if sufficient physical memory exists. Next, it is important to understand capacity limits of the resources you will be using. For example, if the RDBMS resides on a server that is stretched to capacity, SQL query efficiencies may not be enough to help you obtain the desired time savings. If this is the case it might be prudent to consider running the queries when there is less network traffic such as earlier in the morning or later in the day or if possible during the night. The advantage to this approach is that there will be (1) less concurrent users competing for the same limited resources and (2) you will need to worry less about being held hostage by code submitted by programmers who have not yet learned how to write efficient queries. Inefficient queries can easily create a log jam by unnecessarily monopolizing network connections or other stretched system resources. It is always a good idea to check with a system administrator for ideas on when network speeds are at their best. This is especially true in large organizations dependent on centralized servers (e.g. Veterans Health Administration with over ten thousand users) that may have not increased their capacity as quickly as the expansion of their user base.

SAS/ACCESS

SAS/ACCESS is SAS® software that allows for connectivity to various third-party databases and other software. For accessing SQL Server from SAS® on a Microsoft Windows platform, there are three commonly used SAS/ACCESS products. These include (1) SAS/ACCESS Interface to OLEDB (2) SAS/ACCESS Interface to ODBC and the latest product (3) SAS/ACCESS Interface to Microsoft SQL Server. None of these products are sold with the Base SAS® license. SAS/ACCESS Interface to SQL Server offers some performance gains over the other two but it is not intended to be used with other third-party databases and is not covered in this paper. See SAS® online documentation for further details. There is also documentation and excellent papers (see Johnson 2015, Werner 2014) on SAS/ACCESS Interface to ODBC available but this paper is focused on connectivity to Microsoft SQL Server databases using the SAS/ACCESS Interface to OLEDB. OLEDB is a Microsoft API that stands for Object Linking and Embedded Database. Many of the tips included in this paper are relevant for programmers that connect to Microsoft SQL Server using ODBC but the syntax for the CONNECT and LIBNAME statements differs slightly. If you are unsure if you have any of the licensed SAS/ACCESS products, simply run the following code in the SAS® editor and check the SAS® log for the components installed on your PC.

```
PROC SETINIT NOALIAS; RUN;
```

For additional details about the SQL driver installed on your PC see the SAS® technical report entitled *Accessing a Microsoft SQL Server Database from SAS® under Microsoft Windows* (2017).

LIBNAME STATEMENT

SAS/ACCESS offers two different methods for connecting to the SQL Server database. The simplest approach is with a LIBNAME statement. As a refresher in Microsoft SQL Server, data is referenced using a four part layout: [ServerName].[Database].[Schema].[TableorViewName]. In order for SAS® to be able to connect to the table(s) or view(s) of interest in your SQL Server database, it first must know where to look. Fortunately, this can easily be done using a LIBNAME statement just as if you were specifying the Windows folder location for a permanent SAS® data set. The basic generic LIBNAME code looks like the following:

```
libname sqlsrv oledb
init_string="Provider=SQLOLEDB.1; Password=pwd;
Persist Security Info=True; User ID=user; Datasource=sqlserv";
```

Notice in this generic code sqlsrv is the libref that can be used just like a normal two-level qualifier (e.g. work.mydataset, sashelp.class, or in this case sqlsrv.sqltablename) and the data source sqlserv refers to the network server name. Your database administrator (DBA) should be able to help alter this code so that it is specific to your SQL Server database environment. In our case here at VA Connecticut Healthcare, we use Windows NT authentication for connectivity and a typical SAS® LIBNAME statement might look like the following:

```
libname EMR oledb
init_string="Provider=SQLNCL11.0; Integrated Security=sspi;
Persist Security Info=True; Initial Catalog=VACT;
Datasource=VACTserver"; Schema=VACTschema;
```

For this example three of the four parts of a Microsoft SQL Server reference are contained in the LIBNAME. The Datasource VACTserver is the Servername, the Catalog VACT is equivalent to the Database name and the Schema is specified as VACTschema. A schema is just a way to designate the part of the database where some of the tables are stored and where the DBA is willing to grant permission for user access. Note also that EMR is the libref, and SQLNCL11.0 is the specific SQL driver name being used (refer to SAS documentation for steps to determine your computer's driver name).

Having created this LIBNAME, data can now be accessed from the SQL Server database and processed in the standard way in SAS® using either a standard data step or PROC SQL. Data can even be written to a SQL Server table similar to how you might create a permanent SAS® data set assuming that write

permissions have been granted. Notice that additional information about the database schema and a specific catalog may be required and your syntax will differ slightly. Datasource for example will refer to your specific server name, and your DBA will need to grant access to your userid for the specific database schema of interest.

The key point to remember when using the LIBNAME method is that it is better for handling smaller tables. When querying or manipulating larger tables in a SQL Server database (over 1 million rows), either within a data step or within PROC SQL, this method may be constrained by the transfer of data to the SAS® application server. It is also likely the processing done in SAS® would likely be handled more efficiently directly in Microsoft SQL Server.

IMPLICIT SQL PASS-THROUGH

Having created a libref, such as EMR above, it is possible to use this within PROC SQL and have SAS® pass the code to the RDBMS implicitly where it can run directly within the RDBMS rather than doing the processing in SAS®. Depending on how the query is written, processing may occur within the RDBMS or on the SAS® side. SAS® coding conditions as described below will lead to processing which occurs within SAS®. Consequently, this will require the data to be transferred from SQL server to SAS®, thereby negating the advantage of this method when it comes to very large data tables. Some examples include:

1. Using SAS® functions in the SELECT or WHERE clauses.
```
where datepart(datetimevar) > '31DEC1950'D; *SAS Datetime variable;
where upcase(charvar) = 'SMITH'; *Equivalent to T-SQL function upper;
select substr(charvar,1,5) as newvar from EMR.tablename;
```
2. Using SAS® formats in the WHERE clause.
3. Creating a macro variable using the INTO: clause.
4. Using data set options such as DROP or KEEP.
5. Multiple separate librefs; for example one referring to table in the RDBMS and the other referring to a SAS® data set in a permanent or even the WORK library.

Incidentally the option NOIPASSTHRU coded as follows will prevent PROC SQL implicit pass-through.

```
proc sql noipassthru;
```

It is rare this might be wanted but it is possible that in some cases the query optimization will be better run in SAS®. Conversely there are also SQL key words that will trigger implicit pass-through assuming there are no conditions preventing it. These include DISTINCT, JOIN, UNION and COMPUTED (Johnson, 2015).

In the event you are unsure whether your PROC SQL query is being run within the SQL Server database or if it cannot be translated by SAS® for execution there, it can be very beneficial during your testing phase to use the following to get more detailed information in your SAS® log. Once your debugging is complete, it is recommended that you comment out this line of code since it can add time because of the I/O processing it requires.

```
options debug=dbms_select sastrace=',,,d' sastraceloc=saslog nostsuffix;
```

Using this option let us look at a couple of examples to see how the log will look when the SQL pass-through is invoked, and as a comparison how it looks when SAS must bring the data to the SAS server for processing. Both examples rely first on the following LIBNAME:

```
libname BC oledb init_string="provider=SQLNCLI11.0;
         integrated security=sspi; persist security info=true;
         initial catalog=birthcohort; data source=vhaconapp1;" schema=dbo;
```

The first example below does not use the implicit pass-through due to the datepart function:

```
proc sql inobs=1000000;
create table Diagnoses as
select distinct scrssn,admitdatetime,inotpt,icd9code,icd10code,
icdfirst3, datepart(admitdatetime) format mmddyy8. as AdmitDate
from EMR.dx_long_v
where inotpt = 1 and icdfirst3 in ('250' 'E10' 'E11');
quit;
```

A portion of the log message for the above code is the following:

```
DBMS_SELECT: SELECT * FROM "dbo"."dx_long_v"
OLEDB_4: Prepared: on connection 2
SELECT * FROM "dbo"."dx_long_v"
OLEDB: AUTOCOMMIT turned ON for connection id 3
OLEDB:  *-*-*-*-*-*-* COMMIT *-*-*-*-*-*-* on connection 3
OLEDB: AUTOCOMMIT turned OFF for connection id 3
SAS_SQL:  Unable to convert the query to a DBMS specific SQL statement due to an error.
ACCESS ENGINE:  SQL statement was not passed to the DBMS, SAS will do the processing.
DEBUG: DBMS engine returned an error - NO Implicit Passthru.

NOTE: Table WORK.DIAGNOSES created, with 100000 rows and 7 columns.
61     quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           1.24 seconds
      cpu time            1.12 seconds
```

Note for the second example below the variable AdmitDate is no longer created with the SAS datepart function and so the implicit pass-through is invoked and the log message makes it clear this is the case.

```
proc sql inobs=1000000;
create table Diagnoses as
select distinct scrssn, admitdatetime,inotpt,icd9code,icd10code,icdfirst3,
from EMR.dx_long_v
where inotpt = 1 and icdfirst3 in ('250' 'E10' 'E11');
quit;
```

An abridged portion of the log for the second example follows:

```
DEBUG: SQL Implicit Passthru stmt has been prepared successfully.
OLEDB_3: Executed: on connection 3
DEBUG: SQL Implicit Passthru stmt used for fetching data.
ACCESS ENGINE:  SQL statement was passed to the DBMS for fetching data.

NOTE: Table WORK.DIAGNOSES created, with 100000 rows and 6 columns.
54   quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.80 seconds
      cpu time            0.56 seconds
```

EXPLICIT SQL PASS-THROUGH

As noted previously the first and simplest method that SAS/ACCESS can connect to a RDBMS is with the LIBNAME statement. The second approach is done with the explicit SQL Pass-Through Facility and can only be executed within PROC SQL. The explicit Pass-Through Facility involves wrapping the SQL code native to the RDBMS within a CONNECT statement, having it run directly on the RDBMS, and then having the resulting records returned to the SAS® application server. In our case we are interested in the

SQL code native to SQL Server which is known as Transact-SQL or T-SQL for short. Using this method is highly efficient for dealing with large SQL Server tables especially when needing to execute complex queries or when using native SQL code would be beneficial. While the time savings from explicit over implicit SQL pass-through is sometimes negligible, the key point is that both pass-through methods will offer huge run time processing over the traditional data step when working with large data tables.

For programmers that are proficient in T-SQL, the explicit Pass-Through offers the opportunity to leverage this skill to work directly from SAS®. On the other hand, for SAS® programmers, like this author, who are more comfortable using the data step, the advantages offered with the explicit method make learning more about T-SQL well worth the effort. Either way it is best to have a plan for writing the T-SQL that will be passed directly to SQL Server. Ideally if you have Microsoft SQL Server Management Studio (SSMS) available to use, it is preferable to write and test your code within the SSMS. Once code can be successfully run in SSMS it can then be copied and pasted into SAS® and wrapped within syntax that can be used in PROC SQL. While reducing the amount of data to be transferred from SQL Server will generate the bulk of the time savings, it is worth mentioning that SSMS has execution plan tools like the Query Optimizer that can be used to maximize the efficiency of your queries.

Knowing that the explicit SQL Pass-Through is what we want to use how is it done? The basic syntax for usage of SAS/ACCESS Interface to Microsoft® OLEDB within PROC SQL with Windows NT authentication (Note: SQL Server authentication is similar; see the SAS/ACCESS Technical Paper in references for more information) is as follows:

```
proc sql; *Consider NOPRINT option for some data extraction tasks;
  CONNECT to OLEDB as user1

  /* Initialize the connection and specify location of database */
   (datasource=sqlserv provider=SQLOLEDB.1 Integrated Security=SSPI
    Persist Security Info=True; Catalog=SQLdatabasename
    schema= SQLschemaname)

  /* Next piece describes SAS data set with the results of query*/
     create WORK.SASdatasetname as
     Select * from CONNECTION TO user1

  /* Code inside the next parentheses is native T-SQL */
   (Select top 1000 field1, field2
      from [SQLdatabasename].[SQLschemaname].[SQLtablename]
        order by field1);

  DISCONNECT from user1; *Sever connection to remote SQL database;
quit; *Properly close PROC SQL;
```

Note that user1 is a connection alias and is not required. If it is used on the CONNECT statement, then it must also be used on the DISCONNECT statement. Notice also that the connection information that is contained in the first set of parentheses is like what is provided in the LIBNAME method of connecting. Keep in mind this paper is narrowly focused to working with SQL Server and an OLEDB connection. Syntax will vary for other RDBMS products using a compatible SAS/ACCESS product, so it is best to seek out SAS® documentation or check with your local DBA. There are additional options that can be included within the syntax including CURSOR_TYPE, DEFER, UTILCONN_TRANSIENT and READBUFF. READBUFF is an option that has been recommended and is routinely used in our VA environment when accessing our electronic medical record database. READBUFF according to SAS® online documentation "...improves performance by specifying a number of rows that can be held in memory for input into SAS®. Buffering data reads can decrease network activities and increase performance." Our VA DBAs have tried different buffering sizes and have found that READBUFF = 5000 is optimal. The best size for your environment may differ.

Another advantage to the explicit pass-through is the ability to join data from multiple SQL Server database schemas. In this case, the connection is pointing to the database only without the schema reference, and the two schemas (highlighted in yellow) are referenced in the select statement. This can be done using implicit pass-through but is more efficient using explicit pass-through.

```
proc sql;
  connect to oledb as user1
  (provider=SQLncli11 datasource=VACTserver READBUFF=5000
   cursor_type=static utilconn_transient=yes
   PROPERTIES=('Initial Catalog'=VACT 'Integrated Security'=SSPI)
               defer=yes);
  create table ICD9codes as
    select * from connection to user1
 (SELECT a.studyid, a.recdate, a.icd9code, a.inotpt
  from dbo.dx_long_v a inner join
       codes.naaccord_icd9 b
  on a.icd9code = b.icd9code);
quit;
```

Once you have figured out the correct syntax for the connection you may want to store this code within a macro (e.g. named SQLconnection in code below).

```
%macro SQLconnection;
  connect to oledb as user1
  (provider=SQLncli11 datasource=VACTserver READBUFF=5000
   cursor_type=static utilconn_transient=yes
   PROPERTIES=('Initial Catalog'=VACT 'Integrated Security'=SSPI)
               defer=yes);
%mend;
```

Upon compilation of the macro, it can now be referenced whenever needed and save you typing keystrokes.

```
proc sql;
  %SQLconnection;  *Code from the macro above will be inserted here;
     create temp as
     Select * from CONNECTION TO user1
  (T-SQL code here ...)
  Disconnect from user1;
quit;
```

Incidentally a macro can also be employed to store your commonly used LIBNAME librefs. In addition to using macros to save typing, another nice advantage of using the explicit SQL Pass-Through method is that your T-SQL code run from PROC SQL Server can also include calls to stored SQL procedures using the EXECUTE statement. Like SAS® macros, SQL stored procedures are saved for reuse and offer advantages although they have their drawbacks as well. While explaining further goes beyond the scope of this paper, and the knowledge base of its author, it is significant to be aware this functionality exists.

THE PROOF IS IN THE SAS LOG

The best way to really appreciate the benefit of some of the above tips is to try them out and compare the amount of time consumed for the different methods. As has been stressed throughout this paper the biggest gain will be experienced by the data step diehards upon trying out PROC SQL. Both implicit and explicit pass-through will lead to faster code and it will not be unusual to see savings like the following actual examples based on Veterans Affairs medical record data in West Haven CT.

The difference between using a merge with a data step in Example 2a versus the recommended PROC SQL code with an inner join in example 2b results in a savings of five minutes of CPU time.

Example 2a

```
data Datastep_Codes;
merge vacs.dx_long_v (in = a keep = studyid recdate inotpt icd9code)
    codes.naaccord_icd9 (in = b keep = icd9code) ;
by icd9code ;
if a and b; run;
```

Portion of the SAS® log:

```
NOTE: There were 98318184 observations read from the data set VACS.dx_long_v.
NOTE: There were 730 observations read from the data set CODES.naaccord_icd9.
NOTE: The data set WORK.DATASTEP_CODES has 34506179 observations and 4 variables.
NOTE: DATA statement used (Total process time):
      real time          11:20.71
      cpu time           7:29.81
```

Example 2b

```
proc sql;
  create table ProcSql_Codes as
  select a.studyid, a.recdate, a.icd9code, a.inotpt
  from dbo.dx_long_v a inner join
       codes.naaccord_icd9 b
  on a.icd9code = b.icd9code;
quit;
```

Portion of the SAS® log:

```
NOTE: Table WORK.PROCSQL_CODES created, with 34506179 rows and 4 columns.
      quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time          2:31.85
      cpu time           2:26.89
```

The next example came out of a request from a colleague who was struggling with the amount of time his SAS® code was taking to run and was part of the impetus for this paper. As noted earlier the author had also faced a similar challenge when first trying to use a data step for SQL tables with well over 100 million rows. Here was the opportunity to pass along what the author had recently been taught. Notice the huge differential between example 2c that takes over an hour and a half versus example 2d that takes less than five minutes and yields the same results.

Example 2c

```
data CREATININE (drop = LabChemCompleteDateTime
                 rename = (number= testvalue));
set bc_emr.lab_combined_09192018(keep = scrssn patientsid test
                                 LabChemCompleteDateTime number units
where = (strip(upcase(test)) = "CREAT"));
testdate = datepart(LabChemCompleteDateTime); run;
```

Portion of the SAS® log:

```
NOTE: There were 69399165 observations read from the data set EMR.lab_combined.
WHERE STRIP(UPCASE(test))='CREAT';
      quit;
NOTE: The data set WORK.CREATININE has 69399165 observations and 6 variables.
NOTE: DATA statement used (Total process time):
      real time        1:35:20.43
      cpu time         1:32:09.90
```

Example 2d

/*** Improved explicit SQL pass-through code below ***/

```
proc sql;
   connect to oledb as user1
   (provider=SQLncli11 datasource=vhaconapp1 READBUFF=5000
    cursor_type = static utilconn_transient = yes
   PROPERTIES=('Initial Catalog'=birthcohort 'Integrated Security'=SSPI)
        defer = yes);
   create table CREATININE as
   select * from connection to user1
   (select scrssn, patientsid, test, LabChemCompleteDateTime, number, units
    from Birthcohort.emr.lab_combined as a
    where test = 'CREAT');
quit;
```

Portion of the SAS® log:

```
NOTE: Table WORK.CREATININE created, with 69399165 rows and 6 columns.
      quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time          5:19.60
      cpu time           4:50.95
```

The last example shows how using explicit pass-through coding in example 2f can offer a big advantage over PROC SQL code that cannot be executed in the RDBMS but instead must be executed on the SAS application server. This is due primarily because of the datepart function used in example 2e. By using T-SQL in example 2f, the datetime variable Dischargedatetime can be compared to a date value without having to break it up into separate date and time components.

Example 2e

```
Proc sql;
  create table BC_InpatPossible as
  select * From bc_emr.inpat_inpatientdiagnosis
  where datepart(dischargedatetime) gt '30SEP2015'D;
```

Portion of the SAS® log:

```
NOTE: Table WORK.BC_INPATPOSSIBLE created, with 9151905 rows and 12 columns.
NOTE: PROCEDURE SQL used (Total process time):
      real time          8:36.85
      cpu time           8:01.62
```

Example 2f

```
proc sql;
   connect to oledb as user1
   (provider=SQLncli11 datasource=vhaconapp1 READBUFF=5000
    PROPERTIES=('Initial Catalog'=birthcohort 'Integrated Security'=SSPI)
         defer = yes);
    create table BC_Inpat_SQL as
    select * from connection to user1
    (select SCRSSN, Dischargedatetime, ICD10CODE, STA3N
     from Birthcohort.emr.inpat_inpatientdiagnosis as a
    where Dischargedatetime >= '10/01/2015');
```

Portion of the SAS® log:

```
NOTE: Table WORK.BC_INPAT_SQL created, with 9151905 rows and 4 columns.
      disconnect from user1;
      quit;
NOTE: PROCEDURE SQL used (Total process time):
      real time           30.28 seconds
      cpu time            28.31 seconds
```

CONCLUSION

The best way to learn new skills is when you are faced with a task that no longer can be solved with the usual tools in the toolbox. As we know with SAS there are many ways to get the same correct result but not all methods are equally desirable. Selecting the best technique is sometimes a matter of trial and error which can be difficult, especially when a deadline is looming. Fortunately, when it comes to PROC SQL there are many who have already completed those trials before, and applying their techniques is relatively easy. When it comes to large data, making sure it is processed on the RDBMS is the vitally important point. Hopefully this paper will be the impetus to expand your toolbox and become more proficient with T-SQL and SAS PROC SQL so that large data tables will no longer be a problem. Upon trying these methods and then discovering innovative improvements, please be sure to share them with your time crunched SAS® community.

REFERENCES

Crosbie, Stephen. (2013). *Does foo Pass-Through? SQL Coding Methods and Examples using SAS® software.* Paper RF-02-2013, MWSUG 2013. Available at https://www.mwsug.org/proceedings/2013/RF/MWSUG-2013-RF02.pdf

Gupta, Sunil. (2006). *WHERE vs. IF Statements: Knowing the Difference in How and When to Apply.* Paper 238-31, SAS® SUGI 31. Available at https://support.sas.com/resources/papers/proceedings/proceedings/sugi31/238-31.pdf

Johnson, Misty. (2015). *Just passing through… Or are you? Determine when SQL Pass-Through occurs to optimize your queries.* Paper BB-03-2015, MWSUG 2015. Available at https://www.mwsug.org/proceedings/2015/BB/MWSUG-2015-BB-03.pdf

Lafler, Kirk Paul (2017). *Advanced Programming Techniques with PROC SQL*. Paper 930-2017, SAS Global Forum 2017. Available at: https://support.sas.com/resources/papers/proceedings17/0930-2017.pdf

Martin, Kevin. (2017). *Best Practices in Accessing SQL Server Data with SAS.* Available through Veterans Affairs Intranet only at https://vaww.vinci.med.va.gov/vincicentral

SAS Institute Inc. November 2017. SAS Technical Report: Accessing a Microsoft SQL Server Database from SAS® under Microsoft Windows, Content Version 1.1 (replaces TS-265). Cary, NC: SAS Institute Inc. Available at http://support.sas.com/techsup/technote/accessing-microsoft-sql-server-from-sas.pdf

SAS Institute Inc. November 2018. SAS /ACCESS 9.4 for Relational Databases: Reference, 9th Edition. SAS Documentation. Cary, NC: SAS Institute Inc. Available at https://documentation.sas.com/?docsetId=acreldb&docsetTarget=titlepage.htm&docsetVersion=9.4&locale=en

Schacherer, Christopher W. & Detry, Michelle A. (2010). *PROC SQL: From Select to Pass-Through SQL.* Paper at SCSUG 2010. Available at http://scsug.org/SCSUGProceedings/2010/PROC%20SQL%20%20From%20SELECT%20to%20Pass-Through%20SQL%20-%2022OCT2010B.pdf

Schmitz, John (2016). *Solving Common PROC SQL Performance Killers when using ODBC.* Paper BB16, MWSUG 2016. Available at https://www.mwsug.org/proceedings/2016/BB/MWSUG-2016-BB16.pdf

Schreier, Howard. 2008. *PROC SQL by Example: Using SQL within* SAS®. Carey, NC: SASInstitute Inc.

Werner, Nina. (2014). *SAS PASSTHRU to Microsoft SQL Server using ODBC.* Paper SA-03-2014, MWSUG 2014. Available at https://www.mwsug.org/proceedings/2014/SA/MWSUG-2014-SA03.pdf

Williams, Christianna S (2012). Queries, Joins, and Where Clauses, Oh My!! Demystifying PROC SQL. Paper 149-2012, SAS Global Forum, 2012. Available at: http://support.sas.com/resources/papers/proceedings12/149-2012.pdf

ACKNOWLEDGMENTS

Mr. O'Leary would like to thank my talented colleagues, especially Jan Tate and Melissa Skanderson who are always willing to offer technical assistance and support especially as I transitioned from small to large Veterans Health data. Overdue thanks to Christianna Williams who was my first SAS teacher. For too long I avoided the lessons of her paper PROC SQL for Data Step Die-hards but now I can more fully appreciate her pioneering efforts. Shout out also to Peter Charpentier who taught the first SAS class I attended and who has been an invaluable support to me professionally at Yale for over 20 years. He has consistently demonstrated a standard of programming excellence that has inspired his data management team to be more creative and strived to be more proficient.

CONTACT INFORMATION

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

John O'Leary
Veterans Affairs Connecticut Healthcare
950 Campbell Avenue, MZ 151B
West Haven, CT 06419
(203) 932-5711 ext. 2402
john.oleary@yale.edu

APPENDIX: GENERAL TIPS FOR EFFICIENT PROC SQL QUERIES

As noted above, reducing the amount of data that must cross the network from the RDBMS to SAS® is the key goal. In addition to choosing the right method for large data files it is also important to try and write the most efficient SQL code possible. There is no shortage of great resources on the topic of PROC SQL code and a few recommended selections can be found in the references (Lafler 2017, Schreier 2008, Williams 2012). Remembering to write queries that both reduce the number of records and fields selected is a good place to start. Sometimes in our haste it seems simpler to just select all the fields rather being bothered to type out the names of particular fields. For smaller tables this is not a problem but for large data tables this should be avoided. Instead of selecting every field from the source table as shown in Example 1a, it is recommended that you only select the needed fields as demonstrated in Example 1b seen below.

Example 1a

```
proc sql;
  create table all_fields as
  select *
  from source_table
  where DOB > '31DEC1950'D;
quit;
```

Example 1b

```
proc sql;
  create table needed_fields as
  select StudyID, DOB, vitalstatus
  from source_table
  where DOB > '31DEC1950'D;
quit;
```

The efficiency gained from this simple modification will depend on the number and type of fields that have no longer been selected from source_table. Character fields in particular can be rather long so eliminating these from data transmissions across the network is especially important.

Most SAS® programmers understand in a data step that an IF clause requires that SAS® read through every single record in the data set and this can really be inefficient especially for large data sets. One of the big benefits of using a WHERE clause within various SAS® procedures, especially PROC SQL, is that the WHERE condition "is applied before the data enters the input buffer" (Gupta, 2006) thereby subsetting the data for faster processing. It is beyond the scope of this paper to explain the technical details further, but it suffices to say that the WHERE statement is beneficial for both implicit and explicit PROC SQL Pass-Through code.

When using a compound (multiple conditions) WHERE statement within PROC SQL it is a good idea to have the most restrictive condition first. Additionally, if that first condition is an indexed field this will lead to even better results. Let's assume our DBA has indexed the date of birth (DOB) field and that nearly all the records are for living patients. While Example 1c will yield the same results as Example 1d, by reducing the result set first by DOB, Example 1d will run faster.

Example 1c

```
proc sql;
  create table needed_fields as
  select a.StudyID, a.DOB, a.vitalstatus
  from source_table a
  where a.vitalstatus = 'ALIVE' and a.DOB = '31DEC1950'D;
quit;
```

Example 1d

```
proc sql;
  create table needed_fields as
  select a.StudyID, a.DOB, a.vitalstatus
  from source_table a
  where a.DOB = '31DEC1950'D and
        a.vitalstatus = 'ALIVE';
quit;
```