SESUG Paper 266-2018

Proc SQL in Twenty Minutes

John J. Cohen, Advanced Data Concepts, LLC, Newark, DE

ABSTRACT

Proc SQL can be a valuable addition to your SAS® tool set. While the syntax will seem foreign to experienced Data Step programmers, the Structured Query Language can be quite elegant in its own right. Further, unlike a requirement for us to "computer engineer" our Data Step for efficiency, our friends in Cary have built-in intelligence to allow us to submit generic SQL and optimization under the covers does at least some of the rest.

The SQL Procedure makes certain tasks easier (such as Cartesian Products), is a requirement for pulling data from many external databases (to link via "proc SQL pass-through" to link to Oracle, Teradata, MS/Access, and the like), and for many is a preferred tool for common tasks (such as capturing record counts, accessing SAS Dictionary Tables, creating views and indexes, and capturing summary statistics for loading into SAS macro variables).

Finally, in today's complex IT environments, having some familiarity with SQL will allow one to better engage in this broader environment. We will not turn you into an overnight expert, but we will lay a foundation for you to continue to explore and learn on your own. We will introduce the statements, options, and keywords likely to be of immediate interest – the rest is up to you.

INTRODUCTION

SQL, or the Structured Query Language, is an ANSI standard language used widely for retrieving and manipulating data from relational databases and is based on the work of E.J. "Ted" Codd. Proc SQL is SAS' implementation of this language. In the relational terminology, the Data Step **Observations** and **Variables** are typically instead referred to as **Rows** and **Columns**. Instead of **Data Sets**, instead we refer to **Tables**. Beyond that, the following will be the words and clauses which will unlock your access to Proc SQL.

Figure 0 – Words to Live By

- Proc SQL
- Select
- From
- Where
- Into
- Order by
- Group by/having
- As
- Join

We start with the regular keyword **PROC** followed by the procedure name **SQL**.

Proc SQL;

Several options are possible on the **Proc** statement, a few of which will be discussed after building some context. The Proc statement is followed by some number of additional statements.

SELECT/FROM

The **SELECT** statement is used to retrieve data from single or multiple tables. This can consist of a request to return all columns or a subset/list of columns from the supplying table(s). A simple query might look like the following in **Figure 1**. (We will be using the **SASHELP.cars** dataset for most of our examples.)

Figure 1 – Simple Query

```
Proc SQL;
   select *
   from SASHELP.cars;
quit;
```

The asterisk (*) following the **Select** indicates that we wish to retrieve all columns (or variables) from the source table(s). The source table(s) is (are) indicated on the **From** statement. Notice that we end the Proc SQL "step" with a **QUIT** statement (rather than the **Run** statement we use with a **Data Step** or most **PROC**s). The results might resemble something like the partial results in **Figure 2**.

Figure 2 – Simple Query, Partial Results

Make	Model	Type	Origin
Acura	MDX	SUV	Asia
Acura	RSX Type S 2dr	Sedan	Asia
Acura	TSX 4dr	Sedan	Asia
Acura	TL 4dr	Sedan	Asia
Acura	3.5 RL 4dr	Sedan	Asia
Acura	3.5 RL w/Navigation	Sports	Asia
Acura	NSX coupe 2dr manual S	Sports	Asia
Audi	A4 1.8T 4dr	Sedan	Europe
Audi	A41.8T convertible 2dr	Sedan	Europe
Audi	A4 3.0 4dr	Sedan	Europe
Audi	A4 3.0 Quattro 4dr manual	Sedan	Europe
Audi	A4 3.0 Quattro 4dr auto	Sedan	Europe

Notice that the results from the program in **Figure 1** are a report and NOT a table. If we wanted to simply generate summary statistics such as an observation (row) count without a report – likely posting that count to a SAS macro variable, the following in **Figure 3** might be our first try. The **NOPRINT** option on the **Proc SQL** statement will suppress said report.

Figure 3 – Generating a Row Count (first try)

```
Proc SQL NOPRINT;
   select *
   from sashelp.cars;
quit;
      /*** display certain PROC SQL Automatic Macro Variables ***/
%put LOG: SQLOBS=&sqlobs. SQLOOPS=&sqloops. SQLRC=&sqlrc.;

LOG: SQLOBS=1 SQLOOPS=11 SQLRC=0
```

These automatic **Proc SQL** SAS macro variables can be useful in tracing the internal workings of our SQL step in debugging and for performance considerations. Something like this is often employed as an efficient method for capturing a record count from a SAS dataset. Curiously, in this construct, the value of the automatic SAS macro variable – **&SQLOBS**. – was returned as a value of "1". The macro variable SQLOBS "contains the number of rows that were processed by an SQL procedure statement." (Using the PROC SQL Automatic Macro Variables) In this instance, only one row was processed – which is certainly not what we had in mind. Out of scope for an introductory discussion, except to say that **Proc SQL** may NOT process observations sequentially in the way that we are accustomed to with the **Data Step**.

For generating a reliable observation count we look at the example in **Figure 4**. Our result will be stored **into:** macro variable &nobs. Much of the documentation warns that the value returned will represent the number of rows which were processed. If we included a filtering clause (as in **Figure 5**) or some other logic whereby we dropped out of the SQL step **BEFORE** processing all of the incoming rows, the resulting count will no longer represent the total number of observations in the original/source dataset.

Figure 4 – Generating an Observation Count

```
Proc SQL NOPRINT;
   select count(*) format=comma15.0 into :nobs
   from sashelp.cars;
quit;

%put LOG: SQLOBS=&sqlobs. SQLOOPS=&sqloops. SQLRC=&sqlrc.;
%put nobs=&nobs.;
LOG: SQLOBS=1 SQLOOPS=11 SQLRC=0
nobs=  428
```

WHERE

Next we look at a simple filtering expression using a **WHERE** statement (please see **Figure 5**). Here we are looking to extract those observations meeting the condition on the **WHERE** clause, namely that the values of MPG_highway are greater than 40. Notice also that we are using a variety of **SELECT** statements. In both **Figures 1** and **3** we see the following:

```
select *
```

As suggested, this means that SAS will return all of the columns associated with the table(s) in the FROM statement. In **Figure 4**, in contrast, we are instead calculating a value (a statistic), associating a format with the result, and then storing the formatted result "**into** :" a SAS macro variable.

```
select count(*) format=comma15.0 into :nobs
       statistic          format          macro variable
```

Finally, in **Figure 5** below we have a different syntax, listing out individual columns separated by a comma delimiter. We also see an **ORDER BY** statement, indicating that we want our result sorted by the column **MPG_highway** in **DESC**ending order.

```
select make, model, type, MPG_highway
```

Figure 5 – Generate Report with Limited Column list, filtering with a WHERE clause

```
Proc SQL;
   select make, model, type, MPG_highway
   from sashelp.cars
   where MPG_highway > 40
   order by desc MPG_highway;
quit;
```

The resulting report is displayed in **Figure 6**.

Figure 6 - Report

Make	Model	Type	MPG_highway
Honda	Insight 2dr (gas/electric)	Hybrid	66
Toyota	4dr (gas/electric)	Hybrid	51
Honda	Civic Hybrid 4dr manual (gas/electric)	Hybrid	51
Volkswagen	Jetta GLS TDI 4dr	Sedan	46
Honda	Civic HX 2dr	Sedan	44
Toyota	Echo 4dr	Sedan	43
Toyota	Echo 2dr manual	Sedan	43

The **WHERE** statement comes with a rich set of operators, many of which we are familiar from **Data Step** programming. We will look at the **LIKE** operator in conjunction with the "**%**" wildcard. In Figure 7 we are looking for cars (**MODEL**) where the character string "manual" appears someplace within the variable's value.

Figure 7 – LIKE Operator

```
Proc SQL;
   select make, model, horsepower
   from sashelp.cars
   where model like '%manual%'
   order by make, model;
quit;
```

The wildcard before the character search string indicates that we wish to filter for rows where any set of characters can appear before "'manual". (Note that this includes blanks or no characters – the first word in the value can be the search string.) Similarly, the trailing wild card allows for any character string following the search string to be included, again including blanks or no values. Without the trailing "%", the Acura and Honda entries would not have been identified (both have characters after the search string).

Figure 8 – Results of Applying the LIKE Operator

Make	Model	Horsepower
Acura	NSX coupe 2dr manual S	290
Audi	A4 3.0 Quattro 4dr manual	220
Honda	Civic Hybrid 4dr manual (gas/electric)	93
Kia	Rio 4dr manual	104
Lexus	IS 300 4dr manual	215
Mazda	RX-8 4dr manual	238
Toyota	Echo 2dr manual	108

MORE ON SELECT

The **SELECT** statement is enormously powerful and flexible as we will see in the next several examples. In **Figure 9** we show another use of the **SELECT into :** to create a SAS macro variable. In this instance we are summarizing across multiple observations. We use the **DISTINCT** function to select unique values of **MAKE** from **sashelp.CARS** and place this list into a macro variable called **LIST**. We are indicating that we want SAS to include certain characters as delimiters with a **SEPARATED BY** clause, followed by the desired characters within single quotes.

Figure 10 **– SELECTing a List of Values INTO : Our Macro Variable**

```
Title1 'Luxury Car Brands';
Proc SQL;
   select distinct(make) into :list separated by '","'
   from sashelp.cars
   where MSRP >= 75000;   /*** arbitrary cutoff for "luxury" ***/
quit;

%put &list.;
```

The result – displayed in our log by the **%put** statement – includes the requested delimiter characters (double-quote/comma/double-quote):

Acura","Audi","Cadillac","Dodge","Jaguar","Mercedes-Benz","Porsche","Volkswagen

Macro variable **LIST** has been dynamically assigned a value – a different run from a different time period or with a different **WHERE** clause or a different choice on our **SELECT** statement would result in a different list possibly each time – data-driven and hard code-free. To use this we will likely need to add a leading and a trailing double-quote as in **Figure 11**.

Figure 11 – Using Macro Variable List (dynamically-created in Proc SQL step)

```
Libname carsales XLSX 'users/a123456/SESUG2018/care_sales.xlsx';

Title2 'Alternate Brand Name';
Proc Print data=carsales.names;
   where short_name in ("&list.");
Run;
```

For our list – placed in SAS macro variable LIST including certain character delimiters – to be syntactically correct – we must also add a leading and trailing double quote, at least for the usage in our WHERE statement as partially illustrated here, with the resulting report in **Figure 10**:

```
where short_name in ("Acura","Audi",(rest),"Porsche","Volkswagen");
```

The results are displayed in **Figure 12** – a report created in an entirely different step from a different dataset from the source dataset and PROC SQL step. Note that not all makes in our list (created from the sashelp.cars dataset) match up to observations in the alternate dataset (carsales.names).

Figure 12 – Report Generated with Dynamically-Created List in Our WHERE Clause

```
                        Luxury Car Brands
                      Alternate Brand Names
 Obs    Sales_Data_Name                      Short_Name

   1    Audi of America Inc.                 Audi
   2    Jaguar                               Jaguar
   3    Mercedes-Benz                        Mercedes-Benz
   4    Porsche Cars NA Inc.                 Porsche
   5    Volkswagen of America Inc.           Volkswagen
```

GROUP BY/HAVING

Till this point we have been looking at rows individually (save the exception of the **DISTINCT** function which looks across rows) and have employed the WHERE clause to filter on row. We will next look at **GROUP BY** and use the **HAVING** clause in an analogous fashion. This clause causes the observations to be organized within the groupings we request and allows us to calculate summary statistics within each grouping. In **Figure 13** we are **SELECT**ing **MAKE**, along with three statistics with which we are associating column aliases (select new_var as alias_name). We are associating formats and labels with these new columns as appropriate. We are **GROUP**ing **BY MAKE** and then filtering for average prices greater than 45,000.

Figure 13 – Example of GROUP BY/HAVING

```
Proc SQL;
   select make,
          max(MSRP) as top_of_the_line format=dollar8.0,
          avg(MSRP) as typical_price format=dollar8.0 label='mean price',
          count(*) as offerings
   from sashelp.cars
   group by make
   having calculated 3 > 45000
   order by typical_price desc;
quit;
```

Notice that we **ORDER BY** one of the computed variables after our **GROUP BY** and **HAVING** statements. Two special notes regarding the latter, however. We are referring to the third column, typical_price, by the column number ("3"). We also are including a **CALCULATED** qualifier in front of the computed column. The internal ordering of processing in Proc SQL is such that the column may NOT yet have been calculated when that statement is executed. The qualifier – a SAS extension to ANSII Standard SQL – instructs SAS to create the calculated field "in advance". The resulting report is displayed in **Figure 14**. Notice that the associated Dollar formats and the variable label are displayed in the report.

Figure 14 – GROUP BY/HAVING Report Results

make	top_of_the_line	mean price	offerings
Porsche	$192,465	$83,565	7
Jaguar	$86,995	$61,580	12
Mercedes-Benz	$128,420	$60,657	26
Cadillac	$76,200	$50,474	8
Hummer	$49,995	$49,995	1
Land Rover	$72,250	$45,832	3

JOIN

Our final topic will be that of combining datasets by observation. We will contrast an **INNER** and **OUTER JOIN** and will be combining sashelp.cars with a second dataset, carsales.sales, which is displayed in part in **Figure 15**.

Figure 15 – carsales.sales Dataset (partial)

short_name	manufacturer	March_2018_sales_000
GMC	General Motors Corp.	$296,138
Ford	Ford Motor Company	$243,021
Toyota	Toyota Motor Sales USA Inc.	$222,782
Chrysler	Chrysler	$211,943
Nissan	Nissan North America Inc.	$162,535
Honda	American Honda Motor Co Inc.	$142,392
Hyundai	Hyundai Motor America	$61,540
Subaru	Subaru of America Inc.	$58,097
Kia	Kia Motors America Inc.	$50,654

While there is more than one syntax for describing a **JOIN**, we will stick to a single approach for each of these. The **INNER JOIN** will combine those rows which are common to both datasets on the key variable(s). (This is the equivalent of "IF A and B" in Data Step/By/Merge.) We will create table ALIASES so that we can identify from which table a given column originated. Finally, we will describe the JOIN condition using a WHERE statement. This is displayed in **Figure 16**.

Figure 16 – Inner Join Program

```
Proc SQL;
title 'manufacturers offering manual transmission';
   select a.make,
          b.manufacturer,
          b. March_2018_sales_000 label='sales across all models'
   from sashelp.cars a,
        carsales.sales b
   where a.model like '%manual%'
     and a.make = b.short_name      /*** INNER JOIN ***/
   order by a.make;
quit;
```

We refer to columns using a two-part name. Their column name (or column alias as appropriate) and the table alias. Sashelp.cars is assigned a **table alias** of "a" and so we refer to columns from that table as a.column_name. Similarly, the second table is given a **table alias** of "b" and so we refer to columns in that table as b.column_name. We are joining on the columns **a.make** and **b.short_name** – in SQL the column names do **NOT** need to be identical. Notice that there are two **WHERE** conditions. The second is the **JOIN**, while the first is filtering rows coming from table a and is not required to appear on the resulting table **SALES**. The results are displayed in **Figure 17**.

Figure 17 – Inner Join Results

```
manufacturers offering manual transmission
                                          sales across
make                    manufacturer        all models
--------------------------------------------------
Audi        Audi of America Inc.              $20,090
Honda       American Honda Motor Co Inc.     $142,392
Kia         Kia Motors America Inc.           $50,654
Mazda       Mazda Motor of America Inc.       $33,302
Toyota      Toyota Motor Sales USA Inc.      $222,782
```

Notice that as an **INNER JOIN**, our results will include only those rows common to both datasets (who in the sashelp.cars dataset have at least one manual transmission offering.) So Acura and Lexus, which were reported on in **Figure 8**, are **NOT** reflected here. In contrast, an **OUTER JOIN** will allow us to include all common rows **-AND-** depending on our query, as well as some rows appearing only in one or the other table being combined. **Figure 18** displays a LEFT JOIN, where all the rows in common as well as the rows remaining rows in the first ("left") table are returned. This is analogous to the **Data Step/By/Merge** "if a".)

Figure 18 – Left Join Program/ON

```
Proc SQL outobs=10;
   select distinct(a.make) as make,
          b.manufacturer
   from sashelp.cars a
      left join
        carsales.sales b
     on a.make = b.short_name
   order by make;
quit;
```

Three noteworthy items appear in this example. The **JOIN** condition is described on an **ON** statement (rather than the **WHERE** in the prior example of an **INNER JOIN**). We are **ORDER**ing **BY** a column **ALIAS** rather than the original column. Finally, we are limiting the final output using an OUTOBS option on the Proc SQL statement. (In this case, only the first 10 observations meeting the query conditions will be returned.) The results are displayed in **Figure 19**. Included are the rows from sashelp.cars which match up to their counterparts in carsales.sales as well as therows in the former which do **NOT** match to the latter. (These will have blanks as their respective values for the column **manufacturer**.)

Figure 19 – Left Join Results

```
make                    manufacturer
-----------------------------------
Acura
Audi        Audi of America Inc.
BMW         BMW of North America Inc.
Chevrolet
Chrysler    Chrysler
Ford        Ford Motor Company
Honda       American Honda Motor Co Inc.
Infiniti
Isuzu
Jaguar      Jaguar
```

CONCLUSION

Proc SQL is a highly functional, flexible, and powerful tool, sufficiently similar to other instances of SQL to allow considerable crossover in an increasingly multi-tool environment. When pulling data from external Databases (Oracle, Teradata, MySQL, etc.) it may even be required. While the challenges of learning a new programming language are non-trivial, focusing on the key concepts introduced should allow one to become productive in short course.

REFERENCES AND RESOURCES

Agnihotri, Kunal and Borowiak, Ken, 2014, "Tips for Creating SAS®-Based Applications for Oracle Clinical", Proceedings of the PharmaSUG 2014 Conference, San Diego, California. Available at: http://pharmasug.org/proceedings/2014/AD/PharmaSUG-2014-AD19.pdf

Aster, Rick, 2014, **Routine SAS® SQL**, Breakfast Communications Corporation.

Date, C. J., 2000, **An Introduction to Database Systems**, 7th ed., Addison-Wesley.

Jansen, Lex, https://lexjansen.com/

Lafler, Kirk Paul, 127-29: Efficiency Techniques for Beginning PROC SQL Users, www2.sas.com/proceedings/sugi29/127-29.pdf

SAS® 9.4 SQL Procedure User's Guide, Fourth Edition, SAS® Help Center: Syntax: PROC SQL SELECT Statement, SAS® Help Center: Syntax: PROC SQL SELECT Statement

Using the PROC SQL Automatic Macro Variables, SAS® 9.4 SQL Procedure User's Guide, Fourth Edition

Base SAS(R) 9.2 Procedures Guide, "LIKE condition", available at: https://support.sas.com/documentation/cdl/en/proc/61895/HTML/default/viewer.htm#a002473693.htm

SAS Institute, Inc., 2010, "SQL Pass-Through Facility Specifics for Oracle", **SAS/ACCESS(R) 9.2 for Relational Databases: Reference**, Fourth Edition, Cary, NC, SAS Institute, Inc., available at: http://support.sas.com/documentation/cdl/en/acreldb/63647/HTML/default/viewer.htm#a003113595.htm

Schacherer, Christopher W. and Detry, Michelle A., 2010, "PROC SQL: From SELECT to Pass-Through SQL ",Proceedings of the SouthCentral SAS Users Group, Austin, Texas. Available at: http://www.scsug.org/SCSUGProceedings/2010/Schacherer_2/PROC_SQL_%20From_SELECT_to_Pass-Through_SQL.pdf

Wikipedia, "Edgar F. Codd", retrieved 8/20/2016. Available at: https://en.wikipedia.org/wiki/Edgar_F._Codd

Wikipedia, "Database", retrieved 8/31/2016. Available at: https://en.wikipedia.org/wiki/Database

Williams, Christianna, 2015 , "PROC SQL for PROC SUMMARY Stalwarts", Proceedings of SAS Global Forum 2015, Dallas, Texas. Available at: http://support.sas.com/resources/papers/proceedings15/3154-2015.pdf

TRADEMARKS

CONTACT INFORMATION

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

Name:	John J. Cohen
Enterprise:	Advanced Data Concepts, LLC
Work Phone:	(302) 559-2060
E-mail:	jcohen1265@aol.com