Proc SQL in Twenty Minutes, continued

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 **PROCS**). The results might resemble something like the partial results in **Figure 2**.

**Figure 2 – Simple Query, Partial Results**

```
<table>
<thead>
<tr>
<th>Make</th>
<th>Model</th>
<th>Type</th>
<th>Origin</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acura</td>
<td>MDX</td>
<td>SUV</td>
<td>Asia</td>
</tr>
<tr>
<td>Acura</td>
<td>RSX Type S 2dr</td>
<td>Sedan</td>
<td>Asia</td>
</tr>
<tr>
<td>Acura</td>
<td>TSX 4dr</td>
<td>Sedan</td>
<td>Asia</td>
</tr>
<tr>
<td>Acura</td>
<td>TL 4dr</td>
<td>Sedan</td>
<td>Asia</td>
</tr>
<tr>
<td>Acura</td>
<td>3.5 RL 4dr</td>
<td>Sedan</td>
<td>Asia</td>
</tr>
<tr>
<td>Acura</td>
<td>3.5 RL w/Navigate</td>
<td>Sports</td>
<td>Asia</td>
</tr>
<tr>
<td>Acura</td>
<td>NSX coupe 2dr manual S</td>
<td>Sports</td>
<td>Asia</td>
</tr>
<tr>
<td>Audi</td>
<td>A4 1.8T 4dr</td>
<td>Sedan</td>
<td>Europe</td>
</tr>
<tr>
<td>Audi</td>
<td>A4 1.8T convertible 2dr</td>
<td>Sedan</td>
<td>Europe</td>
</tr>
<tr>
<td>Audi</td>
<td>A4 3.0 4dr</td>
<td>Sedan</td>
<td>Europe</td>
</tr>
<tr>
<td>Audi</td>
<td>A4 3.0 Quattro 4dr manual</td>
<td>Sedan</td>
<td>Europe</td>
</tr>
<tr>
<td>Audi</td>
<td>A4 3.0 Quattro 4dr auto</td>
<td>Sedan</td>
<td>Europe</td>
</tr>
</tbody>
</table>
```
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)**

```sas
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**

```sas
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
```

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 DESCending 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

<table>
<thead>
<tr>
<th>Make</th>
<th>Model</th>
<th>Type</th>
<th>MPG_highway</th>
</tr>
</thead>
<tbody>
<tr>
<td>Honda</td>
<td>Insight 2dr (gas/electric)</td>
<td>Hybrid</td>
<td>66</td>
</tr>
<tr>
<td>Toyota</td>
<td>4dr (gas/electric)</td>
<td>Hybrid</td>
<td>51</td>
</tr>
<tr>
<td>Honda</td>
<td>Civic Hybrid 4dr manual (gas/electric)</td>
<td>Hybrid</td>
<td>51</td>
</tr>
<tr>
<td>Volkswagen</td>
<td>Jetta GLS TDI 4dr</td>
<td>Sedan</td>
<td>46</td>
</tr>
<tr>
<td>Honda</td>
<td>Civic HX 2dr</td>
<td>Sedan</td>
<td>44</td>
</tr>
<tr>
<td>Toyota</td>
<td>Echo 4dr</td>
<td>Sedan</td>
<td>43</td>
</tr>
<tr>
<td>Toyota</td>
<td>Echo 2dr manual</td>
<td>Sedan</td>
<td>43</td>
</tr>
</tbody>
</table>
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 wildcard 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**

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

**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

```sas
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)

```sas
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:

```sas
    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).
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 SELECTing 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 GROUPing BY MAKE and then filtering for average prices greater than 45,000.

Figure 13 – Example of GROUP BY/HAVING

```sql
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 ANSI 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.
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 14 – GROUP BY/HAVING Report Results

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

Figure 15 – carsales.sales Dataset (partial)

<table>
<thead>
<tr>
<th>short March_2018_name</th>
<th>manufacturer</th>
<th>sales_000</th>
</tr>
</thead>
<tbody>
<tr>
<td>GMC</td>
<td>General Motors Corp.</td>
<td>$296,138</td>
</tr>
<tr>
<td>Ford</td>
<td>Ford Motor Company</td>
<td>$243,021</td>
</tr>
<tr>
<td>Toyota</td>
<td>Toyota Motor Sales USA Inc.</td>
<td>$222,782</td>
</tr>
<tr>
<td>Chrysler</td>
<td>Chrysler</td>
<td>$211,943</td>
</tr>
<tr>
<td>Nissan</td>
<td>Nissan North America Inc.</td>
<td>$162,535</td>
</tr>
<tr>
<td>Honda</td>
<td>American Honda Motor Co Inc.</td>
<td>$142,392</td>
</tr>
<tr>
<td>Hyundai</td>
<td>Hyundai Motor America</td>
<td>$61,540</td>
</tr>
<tr>
<td>Subaru</td>
<td>Subaru of America Inc.</td>
<td>$58,097</td>
</tr>
<tr>
<td>Kia</td>
<td>Kia Motors America Inc.</td>
<td>$50,654</td>
</tr>
</tbody>
</table>

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.
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.

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 **ORDERing** **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 the rows 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


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


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


TRADEMARKS

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 registered trademarks or trademarks of their respective companies.

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