

Proc SQL vs. the Data Step – Trick or Treat?

Tom Aldrich,

Washington State Health and Recovery Services Administration

Abstract: This paper will describe the transition from the small, simple data sets of college to the more complex world of government data sets with billions and billions of bytes (pronounce ala Carl Sagan). Simple tricks in the DATA step and some PROC's will be seen to translate into juicy treats in PROC SQL. PROC SQL topics will include the basics of SELECT, the use of DISTINCT, the WHERE clause, the ORDER BY clause, the GROUP BY clause, and JOIN's. PROC SQL topics will be introduced from the standpoint of non-PROC SQL tools including the DATA Step, MERGE, PROC SORT, PROC PRINT, and PROC MEANS. Seeing how basic forms of PROC SQL statements map to other aspects of SAS will offer beginning SAS users a quick, alternate path to understanding SQL. Resources for learning SQL will be addressed, and some considerations of working with large data sets will be touched on.

Having worked in academic support for a number of years, I noted during this time that most faculty and students working with data analysis software barely scratched the surface of data manipulation. Most of the academic work involved a quick and dirty data step followed by statistical analysis. The data sets were often small or if they were large, some helpful staff in computer support could prepare small subsets. Transitioning from the academic environment to a government or corporate environment involving databases with hundreds of tables, each with millions of records, can be quite a shock. Using some dialect of SQL to access and manage this data becomes a matter of survival. Adding PROC SQL to your SAS repertoire allows you to write more compact code, run jobs faster, and deftly pull together information from multiple databases.

SQL is an acronym for Structured Query Language. It is a language for queries, i.e. it is a language for querying or asking questions about a data set. In particular, it is a language that pulls information from a data set, and it uses the SELECT command to do so. The SELECT command has a range of clauses, analogous to options in DATA step statements. This paper will provide a gentle introduction to SQL using this correspondence between SQL clauses and DATA step and PROC step options. Each aspect of PROC SQL will be introduced in comparison to what should be more familiar SAS terrain.

Let's start with some SAS basics. There are two major aspects of SAS – the DATA Step and the PROC step. PROC SQL does a bit of both but you usually work with PROC SQL on data sets that have already been created. Creating a data set from scratch in PROC SQL would be inefficient (see for example Dickstein and Pass, p 3). From here on we will assume that we're working with an existing data set called claimsdata.

One of the first tasks in working with a data set is getting a feel for the data. To do this you have to look at it, as so:

```
proc print data = claimsdata noobs;  
  var monthx cl_id_num reimbursement_amount;  
  where monthx = 10;  
run;
```

As you can see, this PROC would print out three variables for a particular selected value of monthx, without observation numbers. One of the ways to start using PROC SQL is to use it in simple situations like this and compare the results to alternative code as above. The equivalent PROC SQL statements would be.

```
proc sql;
  select monthx, cl_id_num, reimbursement_amount
  from claimsdata
  where monthx = 10;
quit;
```

Looking at the PROC SQL code, you can see that the SELECT statement first names fields that you want to see, as the VAR statement does in PROC PRINT. However, you'll note that in PROC SQL, the field names must be separated by commas. The FROM clause in SQL names the data set you're working with, as the DATA= option does in the PROC PRINT statement. Finally, the WHERE clause in PROC SQL is analogous to the WHERE statement in PROC PRINT. The mapping of PROC SQL to PROC PRINT is

feature	PROC PRINT	PROC SQL
naming data	DATA=d1	FROM d1
naming variables	VAR v1 v2 v3	SELECT v1,v2,v3
applying conditions	WHERE c1	WHERE c1
exiting/running	RUN	QUIT

A natural question when first confronting PROC SQL is what happened to all the semicolons? Is that a trick or a treat? Before you suspect fowl play, a key point here is that the SELECT statement is one command and the features of the SELECT become clauses that you add to the command. One command, one semicolon. You can have multiple selects in one PROC SQL; quit terminates the procedure.

Sometimes when you do a print, things don't print out the way you'd like. Perhaps you'd like to order the listing by some variable, such as cl_id_num (client id number) so that you'll see all the records for one cl_id_num, then all the records for another (greater or lesser) cl_id_num and so on. Then your code would look something like this, where the added code is in italics. From this point on, the RUN and QUIT commands will be omitted.

```
proc sort data= claimsdata;
  by cl_id_num;
proc print data = claimsdata noobs;
  var monthx cl_id_num reimbursement_amount ;
  where monthx = 10;
```

In PROC SQL, you simply add an ORDER BY clause to order the listing by cl_id_num, the by-variable of the SORT procedure.

```
proc sql;
  select monthx, cl_id_num, reimbursement_amount
  from claimsdata
  where monthx = 10
  order by cl_id_num;
```

Notice PROC SQL is already allowing you to produce more simplified, compact code. Now that's a treat. Also, PROC SQL can be more efficient. In this case, we used .29 second of CPU time in the traditional way with PROC SORT and PROC PRINT and .20 seconds of CPU time in PROC SQL. There's another treat.

Once we've looked at our data at the ground level, we can grab our brooms and start moving up to a treetop view and get some summary statistics. What if we wanted to see the reimbursement amounts for each client, summed over all the records? One approach would be to use PROC SUMMARY or PROC MEANS:

```
proc sort data=claimsdata;
  by cl_id_num;
proc means data=claimsdata noprint;
  var total_reimb_amt;
  output out=paydirt sum=amt_paid_claims;
  by cl_id_num; run;
proc print noobs;
  var cl_id_num amt_paid_claims;
```

This code relies on PROC MEANS to generate summary statistics. The summary statistics are to be presented for each `cl_id_num`, hence the BY statement in PROC MEANS and the PROC SORT that precedes it. The PROC MEANS procedure sums up all the values of `total_reimb_amt` for each client and places the sum into a variable called `amt_paid_claims`. Just to make sure you understand what this does, here's some of the output:

<code>cl_id_num</code>	<code>amt_paid_claims</code>
050497386	1746.48
050501384	1450.98
050513621	381.51

The result of these three procedures is a list of clients and the sum of all the reimbursement amounts for each client. These eight lines of code that take .19 cpu seconds can be replaced with the following four lines of code that take .12 cpu seconds.

```
proc sql;
  select cl_id_num, sum(total_reimb_amt) as amt_paid_claims
  from claimsdata
  group by cl_id_num;
```

These statements produce virtually identical output as above, i.e. a list of clients and the sum of all the reimbursement amounts for each client. Where PROC MEANS used a BY statement to specify a BY variable, PROC SQL uses a GROUP BY statement to specify a BY variable. Where PROC MEANS used a VAR statement and a SUM=option to specify the variable to sum and the variable to hold the sum,

```
var total_reimb_amt;
output out=paydirt sum=amt_paid_claims;
```

in PROC SQL you simply specify

```
sum(total_reimb_amt) as amt_paid_claims
```

as a variable name in the SELECT statement.

So far, you can see that PROC SQL provides what can be a simpler and faster vehicle for generating simple reports than working solely with other procedures. You can also use PROC SQL to get frequency counts, averages, minimums and maximums. But the true power of PROC SQL comes into play when generating reports that arise from multiple data sets that must be merged. To see how PROC SQL works in this context, consider a simple merge.

We might want to look at claims data for a select group, say all the clients that appear in the data set called select_for_months. We can do this with the data step as follows:

```
proc sort data=claimsdata;
  by ClientID;
proc sort data=select_for_months;
  by ClientID;
data selected_claims;
  merge claimsdata (in=A) select_for_months (in=B);
  by ClientID;
  if A and B;
run;
```

Consider data in claimsdata and select_for_months as below

claimsdata		select_for_months	
Client ID	reimbursement_ amount	Client ID	code
1	10	1	A
2	30	2	A
3	40	4	B

After the merge is run, we get the following output:

selected_claims		
Client ID	reimbursement_ amount	code
1	10	A
2	30	A

As you know, the IN=A and IN=B options in conjunction with the IF A AND B statement direct the data step to only include records where ClientID is in both files. These eight statements can be replaced with the following two statements, although if you count the clauses of the PROC SQL, you have four statements/clauses. To simplify the resulting comparison, we'll work with

data sets that do not have duplicate ClientID's. Whereas the combination of PROC SORT's and MERGE requires .15 cpu seconds, PROC SQL uses only .03 seconds.

```
proc sql;  
  create table selected_claims as  
  select a.*,b.code  
    from claimsdata as a , select_for_months as b  
   where a.ClientID = b.ClientID;  
quit;
```

This code shows a few aspects of SQL new to this paper. The first new aspect is CREATE TABLE. Where the data step uses

```
data selected_claims;
```

to tell SAS it is creating a new file called selected_claims, PROC SQL uses CREATE TABLE:

```
create table selected_claims as
```

What's a table? As far as SAS is concerned, it's the same as a data set. That is, when you specify a table to create in PROC SQL, it creates a data set and when you reference a table in PROC SQL, SAS assumes you are referencing a data set. They're interchangeable, but the language of SQL works with tables, so within PROC SQL data sets are referred to as tables. The correspondence works with aspects of the table as well. A row within a table corresponds to an observation within a SAS data set. A field or column within a table corresponds to a variable in a SAS data set. As an indication of the equivalence between a table and a data set, you can use data set options with tables, as below

```
from claimsdata (obs=10)
```

which can be useful for testing.

Another feature of the SELECT statement is the use of an alias. When you say

```
from claimsdata as a
```

you are assigning an alias to the data set `work.claimsdata` so that you can reference it with the alias. Another feature of the statement is the use of *, a wild-card character standing for all fields in the table. Thus a.* means all the fields, or variables, in work.claimsdata. You'll also note that in PROC SQL you can list more than one table in the FROM statement and you must separate the tables with commas. PROC SQL likes commas; SAS does not. Have fun going back and forth. That is more trick than treat. If your program bombs on syntax, check the commas.

Now for the juiciest detail of PROC SQL; it's in the where clause. In the data step we said

```
merge claimsdata (in=A)  
  select_for_months (in=B);  
by ClientID;
```

and the equivalent statements in PROC SQL were

```
select a.*,b.code
      from claimsdata as a, select_for_months as b
     where a.ClientID = b.ClientID;
```

The SELECT statement produces the same result, but the SELECT statement actually says in a clear manner what it is doing. What a treat. The options IN= and the statement IF A AND B filter the output data set to only those records with client id's in A that match client id's in B. PROC SQL generates the same resulting data set, but understanding it is a little easier because the language is more intuitive. You are selecting records from two data sets and the specification for the merge is built into the WHERE statement, so that you are selecting records *where* the values for client id are the same in the two different data sets.

So far, you can see that PROC SQL can provide a more elegant, as in fewer statements, approach to working with data than can be obtained with a combination of the data step and PROC SORT, PRINT, and MEANS. Moreover, PROC SQL can decrease execution times as measured in CPU usage. While these are just small examples, they illustrate what others observe as a general rule: PROC SQL provides a more elegant and efficient alternative to other SAS techniques. Now let's delve into PROC SQL just a little deeper.

You've seen that one alternative to a merge is provided by this statement:

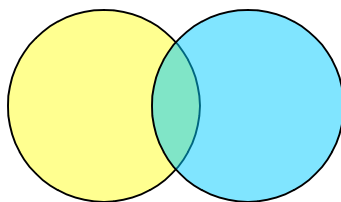
```
select a.*,b.code
      from claimsdata as a, select_for_months as b
     where a.ClientID = b.ClientID;
```

An alternative way to state this in PROC SQL is:

```
select a.*,b.code
      from claimsdata as a join select_for_months as b
     on a.ClientID = b.ClientID;
```

In this formulation, the JOIN is like the MERGE statement in the data step and the ON statement is like the WHERE statement in the SELECT statement in PROC SQL. This is the form you'll most often see in SQL statements, and you'll usually hear the verb join as an alternative to the verb merge when talking about tables. Working with joins also allows you to tap alternative versions of joins that you'll often hear about or see in code. The main types of joins are inner join, left join, right join and full (aka outer) join.

An inner join is the join that we just covered above. For a record to appear in the output data set, the corresponding record as determined by the on fields must appear in each of the input data sets. It is thus like an intersection, as in the green area below.



An alternative to the inner join is a left join, where every record in the table on the left of the join appears in the output data set, along with the corresponding fields from the table on the right of the join. This yields an output data set corresponding to both the yellow and green areas above. The SELECT statement for a left join in our example would be

```
select a.*,b.code
      from claimsdata as a left join select_for_months as b
      on a.ClientID = b.ClientID;
```

The output data set would look like this:

Client ID	reimbursement_ amount	code
1	10	A
2	30	A
3	40	

and the equivalent data step, assuming the input data sets are already sorted on the by variable, would be

```
data selected_claims;
  merge claimsdata (in=A) select_for_months (in=B);
  by ClientID;
  if A;
```

The right join, which is just the flip side of a left join would be coded in PROC SQL as

```
select b.ClientID,a.reimbursement_amount,b.code
      from claimsdata as a right join select_for_months as b
      on a.ClientID = b.ClientID;
```

This would correspond to the area of the diagram that is blue and green. The output data set would look like this:

Client ID	reimbursement_ amount	code
1	10	A
2	30	A
4	.	B

The right join would be equivalent to a data step with the following code:

```
data selected_claims;
  merge claimsdata (in=A) select_for_months (in=B);
  by ClientID;
  if B;
```

Finally, a full (aka outer) join would include records in the output data set corresponding to each record in the input data set. One way of coding this in SQL is

```
select *
  from claimsdata as a full join select_for_months as b
    on a.ClientID = b.ClientID;
```

The output data set would look like this

ClientID	reimbursement_ amount	ClientID	code
1	10	1	A
2	30	2	A
3	40	.	
.	.	4	B

The corresponding data step code would be.

```
data selected_claims;
  merge claimsdata select_for_months;
  by ClientID;
```

With the following output:

Client ID	reimbursement_ amount	code
1	10	A
2	30	A
3	40	
4	.	B

In this case, the IF statement is eliminated so that all records from each input data set are included in the output data set. You'll also notice the output is cleaner than with the SQL SELECT statement used above. You can get around this in SQL with the coalesce function. See Foley (2005) for details and a more comprehensive comparison of SQL with various merges.

I mentioned above that we were assuming no duplicate values in the by variable. When this is not the case, you tend to get a Cartesian product of all possible values with PROC SQL, while merge matches records with duplicate by values in sequence. See Foley (2005) for details and a good, simple example. One way of removing duplicates, if that is what you truly need to do, is to use the nodupkey feature of PROC SORT.

```
proc sort data=claimsdata nodupkey;
  by ClientID;
```

This creates an output data set with the same name as the input data set and with all the variables in the original data set. The output data set contains records corresponding to the first occurrence of each unique value of ClientID. The corresponding feature of PROC SQL is the distinct option of the SELECT statement:

```
proc sql;
    create table claimsdata as
        select distinct ClientID from claimsdata;
```

This creates an output dataset with a different name from the input data step and it only contains one variable. Listing more than one variable results in choosing unique variable combinations. A variable list ClientID, StartDate would choose unique combinations of ClientID and StartDate.

Another valuable feature of PROC SQL is that you can work with databases, such as in Oracle, with what is called a pass-through facility. With a pass-through facility you can, with appropriate privileges, pass a series of commands to a database system and then capture the result of those commands in a table. Here is an example.

```
Proc sql;
    connect to odbc (dsn='client' uid=gingrich pwd=eyeofnewt);
    Create table claimsdata as
    Select * from connection to odbc
        (select * from db3.claims where contract = 'America');
```

In this example the SQL code within the *connection to odbc()* clause is passed to the database for remote processing. The SQL code thus has to conform to the guidelines of the SQL for the remote system. Another point is that while you can use the SAS options RENAME=, KEEP=, DROP=, and OBS= in a SELECT statement, they will most likely not work on the remote, non-SAS, system. The best approach here is to work with options not for the SELECT statement but for the PROC SQL statement, for example substituting noobs= in the PROC SQL statement in place of the obs= in the SELECT statement.

Before you get the impression that PROC SQL is always either the most efficient option in terms of CPU time or the most parsimonious option in terms of lines of code, consider this simple data step, example courtesy of Dickstein and Pass (2004, p6).

```
Data table3;
    Set table1 table2; run;
```

The PROC SQL approach to this uses the union keyword. (the corresponding option ensures that the fields with the same name from different tables are appended properly.)

```
Proc sql;
    Create table table3 as
        Select * from table1
            Outer union corresponding
        Select * from table2;
```

Not only is the sql code less parsimonious than the data step code, but the resultant cpu time was .17 seconds vs. .09 seconds for the data step in an example I ran. (PROC Append took .07 seconds to append table2 to table1.) Hmm, more trick than treat?

A few data tricks will close out this paper. First, you can transform data in PROC SQL by entering the formula in the SELECT command. For example, rather than saying

```
d = b/c;
```

In a data step, you could include the formula as follows

```
Select a, b/c as d
```

The as clause comes in handy for renaming variables, e.g.

```
select y2n as income_to_needs
```

to both select variable y2n and rename it as income_to_needs. Standard naming conventions of SAS apply to the as clause.

Second, you not only have relational operators available for logical conditions but you have fuzzy logic available as well. Using the wildcard character %, which is a stand-in for an indefinite number of characters, the clause

```
where denial_reason like '%incomplete%'
```

would find all records where denial_reason contained the substring 'incomplete' anywhere in denial_reason. It would thus find each of the following:

```
incomplete patient information  
patient information is incomplete
```

This and other useful comparisons of PROC SQL with the data step are available in the excellent and more advanced paper by Williams (1999). Additional recommended sources for further study are the introductory paper by Pete Lund (2005), the SAS® 9.1 SQL Procedure User's Guide, and the book by Prairie (2005), listed in increasing order of difficulty.

Acknowledgements: The author wishes to thank Beth Drake of The Washington Institute for Public Policy, David Fichtenberg of the Washington Department of Social and Health Services, and Keri Jetzer of the Washington State Department of Corrections for helpful review and comments on this paper.

Misc. Details.

1. Comments within PROC SQL work as elsewhere.
2. Naming conventions work the same within PROC SQL, except that when referencing tables in remote database systems, the naming conventions of the system must be followed.
3. You can include formats in the SELECT statement, e.g.

```
select reimbursement format = dollar12.2
```

4. You can use title statements to add title lines to your output for PROC SQL.

References

- Dickstein, Craig and Pass, Ray. "Data Step vs. PROC SQL: What's a Neophyte to Do?" *Proceedings of the Twenty-Ninth Annual SAS Users Group International Conference*. Cary, NC. SAS Institute, Inc. May 2004.
<<http://www2.sas.com/proceedings/sugi29/269-29.pdf>> (Accessed 6/29/2006).
- Foley, Malachy J "MERGING vs. JOINING: Comparing the DATA Step with SQL" 2005 *Midwest SAS Users Group Conference*. October 2005.
<<http://www.mwsug.org/Conference/2005/Tutorials/TS800.pdf>> (Accessed 7/12/2006).
- Lund, Pete "An Introduction to SQL in SAS" *Proceedings of the 2005 Pacific Northwest SAS Users Group Conference*. October 2005.
< http://www.pnwsug.com/Conference_2005/Proceedings/Papers/Pete%20Lund%20-%20An%20Introduction%20to%20SQL%20in%20SAS.pdf> (Accessed 7/24/2006).
- Prairie, Katherine. 2005. *The Essential Proc SQL Handbook for SAS Users*. Cary, NC: SAS Institute Inc.
- Williams, Christianna S.. "PROC SQL for DATA Step Die-Hards" *Proceedings of the Twelfth Annual Northeast SAS Users Group Conference*. August 1999
<<http://www.nesug.org/html/Proceedings/nesug99/ad/ad121.pdf>> (Accessed 7/12/2006).