When To Use or Not Use SQL?
A SAS Coders Perspective.

Don Boudreaux, Ph.D.,
SAS Institute Inc., Austin, TX.

ABSTRACT

SAS provides a very rich collection of programming capabilities for its users. There are lots of functions, call routines, formats, and data step coding structures. SAS has a large collection of procedures for processing data. SAS also has a number of language interfaces and embedded languages. SQL is one of these embedded languages and for many years now SAS coders have been able to use PROC SQL to utilize this tool. The question then arises: when to use SQL and when to avoid it? This paper will attempt provide some practical suggestions and examples of both.

INTRODUCTION

SQL is an embedded language that has been available to SAS programmers from within PROC SQL since version 6.06 in 1989. It is sometimes my language of choice for a given task and sometimes it is a tool I wouldn't even attempt to use. Presented below are a number of examples of both situations.

WHEN NOT TO USE SQL

There are a number of things that SQL was never designed to do including: advanced data analysis, graphics, matrix manipulation, text mining, and data mining. Although, it may be cheating to mention these as they are typically tasks that are done within SAS procedures. It does help focus the rest of our discussion into a more limited perspective of SQL coding vs. "traditional" data step programming.

[1.] SQL cannot read text files.

SQL was designed to be a language that would primarily query and manage existing database files. Reading text files is a bit beyond that prevue. Now, to be fair, if I am dealing with small amounts of data to enter into a new file then SQL does have a rudimentary set of instructions for defining a dataset structure and loading it with records. Consider the following example:

```
proc SQL ;
create table WORK.DFWLAX
(  Flight  char(3),
   Date char(8),
   Dest char(3),
   FirstClass float,
   Economy float )
;
insert into WORK.DFWLAX
   values ('439', '12/11/00','LAX', 20, 137)
   values ('921', '12/11/00','DFW', 20, 131)
   values ('114', '12/12/00','LAX', 15, 170)
   values ('982', '12/12/00','dfw',  5,  85)
   values ('439', '12/12/00','LAX', 14, 196)
;
quit ;
```

This code uses a CREATE statement to define a table (SQL terminology for a dataset) and an INSERT statement loads 5 records (SQL terminology for observations). This is a very similar scenario to reading data in a data step with an INPUT statement and DATALINES. However, while this is fine for very small datasets, like classroom examples, it obviously wouldn't be very useful for loading in a large amount of information. With the data step we can just pitch the DATALINES and start coding an INFILE statement. In SQL you are just out of luck. The suggestion here is DON'T even go there - just use a data step.

[2.] SQL cannot write text files.

Just as SQL cannot read text files it also cannot write text files. Now this is not to say that SQL output cannot be either saved to or redirected into a text file - it can. What I am saying is that the complex types of custom report writing that SAS data step programmers are use to is unavailable within the syntax of SQL. There is no FIRST., LAST., _N_, OUTPUT, or END= available in SQL. And, while I might be able to mimic a very simple customized report, the best suggestion here is again DON'T.

[3.] SQL cannot create multiple datasets at the same time.

Consider the following example where I am building two tables:

```
proc SQL ;
create table WORK.LA_FLIGHTS as
select
   *,
   sum(FirstClass, Economy) as Total
from
   IA.DFWLAX
where
   Dest="LAX"
;
create table WORK.DALLAS_FLIGHTS as
select
   *,
   sum(FirstClass, Economy) as Total
from
   IA.DFWLAX
where
   Dest="DFW"
;
quit ;
```

Note that the SQL code contains two CREATE statements - one per new table. In the data step I can create as many new datasets as needed within a single data step.

```
data  WORK.LA_FLIGHTS
      WORK.DALLAS_FLIGHTS ;
   set IA.DFWLAX ;
   Total=sum(FirstClass, Economy) ;
   select (Dest) ;
      when ("LAX") output WORK.LA_FLIGHTS ;
      when ("DFW") output WORK.DALLAS_FLIGHTS ;
      otherwise ;
   end ;
run ;
```

The data step allows multiple OUTPUT statements to control the writing of data values to any one or multiple datasets. The data step also allows an unlimited number of datasets to be open or written (within the constrains of available memory). If we are talking one or two datasets then either language is fine. But, if I am dealing with simultaneously building more than a few datasets at a time then SQL is clearly not the preferred tool.

[4.] SQL has no capability to transpose data.

I mention this specifically as it is a task that I occasionally need to do. I typically use a data step with ARRAYs and explicit OUTPUT statements to accomplish this. I would also have the ability to do this type of processing in PROC TRANSPOSE or PROC IML (interactive matrix language). But as array processing and output control (or anything similar) are not part of that language, what I would not have as an option here is SQL.

[5.] SQL has no capability for BY processing.

Consider the simple PROC SORT and PROC PRINT shown below. The output that would be generated would essentially give me a listing for each different JobCode value, all shown within a single report.

```
proc sort
   data=ia.empdata
   out=work.empdata ;
   by JobCode ;
run ;
proc print data=work.empdata ;
   var JobCode EmpID Salary ;
   by JobCode ;
   sum Salary ;
run ;
```

Now, in SQL there is nothing analogous to BY processing in Proc or Data Step coding. The closest that SQL could come to the results would be to generate group reports using WHERE clauses and appending them with OUTER UNION :

```
proc SQL ;
select  JobCode, EmpID, Salary
from    IA.EMPDATA
where   JobCode='FLTAT'
outer union corr
select  sum(Salary) as Salary
from    IA.EMPDATA
where   JobCode='FLTAT'
outer union corr
select  JobCode, EmpID, Salary
from    IA.EMPDATA
where   JobCode='PILOT'
outer union corr
select  sum(Salary) as Salary
from    IA.EMPDATA
where   JobCode='PILOT'
outer union corr
select  sum(Salary) as Salary
from    IA.EMPDATA
;
quit ;
```

This is especially fun given that the PROC PRINT results included sums that also have to be appended in. Note that if the PRINT had requested a PAGEBY this could also be mimicked - just remove the OUTER UNION CORR instruction between different JobCode groups. Anyway, this type of coding is "cute" as a classroom exercise, but untenable as production code. In this situation, the suggestion is again - do not use SQL for this type of task.

[6.] SQL should not be used for algorithmic programming.

The order of which instructions within an SQL query are done is highly dependent upon the background optimizer. I would feel comfortable with the idea that the WHERE clause processing in a query is done first and the ORDER BY is done last. But, I would not want to code an algorithium that is dependent on processing order with SQL. In general, the more of a "programming" task I have the less likely I am to use SQL. SQL does not support the following structures, statements, and options that I commonly need in data step coding: ARRAYS, DO LOOPs, HASH TABLES, OUTPUT statements, FIRST. and LAST. processing, _N_, END=, and SAS variable lists (x--z , x1-x3 , x: , _all_).

WHEN TO USE SQL

Having mentioned a few things that SQL cannot do easily (or at all) should not be taken to mean that the language should not be used. Every language has strengths and weaknesses. As a matter of fact, there are a number of things that I highly prefer to use SQL for. Consider the following:

[1.] SQL can easily remove duplicates.

With SQL there is a very simple syntax for building a new dataset that has no duplicate observations.

```
proc sql ;
create table WORK.NODUPS as
select
   distinct *
from
   WORK.SOURCE
;
quit ;
```

This is not to imply that removing duplicates in SQL is more efficient than using a more tradition approach with PROC SORT. It just seems to be a very concise and easy to remember syntax.

[2.] SQL is good at remerging summary statistics.

Consider the task of taking a measurement and comparing it the same variable's overall mean (or any other simple statistic). One scenario for accomplishing this would be to run the original data through PROC MEANS and create a dataset that contains the mean of the variable of interest. Then use a data step to remerge this value back in with the original data:

```
proc means data=WORK.SOURCE noprint ;
   var x ;
   output out=WORK.MEAN mean=xMean ;
run ;

data WORK.USEMEAN ;
   if _n_=1 then set WORK.MEAN(keep=xMean) ;
   set WORK.SOURCE ;
   diff=x-xMean ;
run ;
```

With SQL there is a very simple syntax for doing the same thing in a single request. If the native SQL summary functions are used, the generated statistic is automatically remerged into the result set and immediately available as a new column or for use in other calculations. The same would be true for group level statistics.

```
proc sql ;
create table WORK.USEAVG as
select
   x,
   x-avg(x) as diff
from
   WORK.SOURCE
;
quit ;
```

However, there is a caveat that needs to be mentioned here. There are native SQL functions for the mean (mean() or avg()), minimum (min()), maximum (max()), standard deviation (std ()), and count (count() or n()). There are no advanced statistical functions - the median comes to mind here. The suggestion here is to stick with SQL for simple statistics and move to a proc / data step approach for complex ones.

[3.] SQL can easily build a macro variable loaded with a unique list.

This would have to be my favorite SQL trick - I need a macro variable loaded with a unique list of values. I may want to use the value list in a TITLE or in a WHERE statement or in a macro program loop to generate custom code. In any case, it is very simple to get this generated within an SQL query. Consider that I have a dataset (WORK.SALES) with thousands of sales transactions coming out of four store sites (Site): Austin, Boston, Dallas, and Seattle. Now, I want a "data driven" query that would load the unique site values into a single macro variable for subsequent use in a SAS program. The SQL code to do this is as follows:

```
proc sql ;
select
   distinct Site
into
   :mv_Site separated by ", "
from
   WORK.SALES
;
quit ;
%put &mv_Site ;
```

The macro level PUT statement following the query could then be used to display all of the distinct values into the log and show us:

```
AUSTIN, BOSTON, DALLAS, SEATTLE
```

Again we see the use of the DISTINCT keyword to generate unique values. The INTO can be viewed as an SQL version of the CALL SYMPUT routine and the SEPARATED BY provided the comma and space between the data values in the list. Note that this also can be done without SQL, but not nearly as easily.

[4.] SQL can easily build a set of macro variables.

In the previous example a single macro variable was built with a list of value in it. What if I had wanted each of the unique values in its own new macro variable? Again, with SQL there is a simple syntax to generated as well:

```
proc sql ;
select
   distinct Site
into
   :mv_Site1 - :mv_Site4
from
   WORK.SALES
;
quit ;
%put _user_ ;
```

The macro level PUT statement following the query could then be used to display all of the user defined macro variables into the log including:

```
GLOBAL MV_SITE1 AUSTIN
GLOBAL MV_SITE3 DALLAS
GLOBAL MV_SITE2 BOSTON
GLOBAL MV_SITE4 SEATTLE
```

A nice result for so little code. I would also mention that the second macro variable associated with the INTO determines how many macro variables could be created. If there had only been three sites then only three macro variables would have been built. To insure that every value would get put into a macro variable there are two alternatives. Use a large number in the code (like `:mv_site400`) or create a macro variable with a count of how many values exist in the data and use it in subsequent code that would load the macros. The second suggestion is shown below:

```
proc sql ;
create table WORK.DIST_SITES as
select
   distinct Site
from
   WORK.SALES
;

select
   count(*) as n
into
   :n
from
   WORK.DIST_SITES
;
%let n=&n ;

select
   Site
into
   :mv_Site1 - :mv_Site&n
from
   WORK.DIST_SITES
;
quit ;
```

Note that INTO, like a CALL SYMPUT, will take a number and automatically convert it to a character using a right aligned BEST12. representation of the numeric value. This almost always results in having leading blanks, which would not be good when mv_Site&n resolves. A common solution is to reassign a macro variable back into itself without any leading (or trailing) blanks - which is a default behavior of the %LET statement.

[5.] SQL joins build Cartesian products.

Consider the following two example datasets.

```
WORK.NAMES          WORK.TRANS

Id   Name           Id   Deposit
--   ----           --   -------
 1    A              1    10
 1    B              1    11
                     1    12
```

If the two different names for Id in WORK.NAMES represent a name change for a person then I would want the Name "A" to be associated with 10,11,12 and the Name "B" also associated with 10,11,12. An SQL inner join would do just that. The SQL code:

```
proc sql ;
select
   N.Id,
   Name,
   Deposit
from
   WORK.NAMES as N,
   WORK.TRANS as T
where
   N.Id=T.Id
order by
   Name,
   Deposit
;
quit;
```

would generate the following result set. It is a Cartesian product of both names associated with a complete set of the Deposit values.

```
Id   Name  Deposit
--   ----  -------
 1    A     10
 1    A     11
 1    A     12
 1    B     10
 1    B     11
 1    B     12
```

Put into a dataset and used within a system that would allow a look-up based upon name values, this would be great data. However care must be taken with how this data is used, in a different application scenario, a simple summation of the Deposit values would be counting the 10,11, and 12 values twice!

[6.] SQL allows me to display a variable with different formats.

Consider the following example that would list out the variable StartDate using several different formats at the same time.

```
proc sql ;
select
   StartDate format=mmddyy8.,
   StartDate format=date9.,
   StartDate format=worddate.
from
   WORK.EMPLOYEE
;
quit;
```

And though I do not use this a lot, when I need this type of output, the SQL code seems to be a bit easier than the PROC REPORT equivalent.

CONCLUSION

As I have mentioned before, SQL is sometimes my language of choice for a given task and sometimes it is a tool I wouldn't even attempt to use. Like all data processing languages it has its strengths and weaknesses. The trick then is to figure out what those strengths and weaknesses are. In my case, it is a tool I typically avoid when I need to:

i. read text files
ii. write text files
iii. create multiple datasets at the same time
iv. transpose data
v. do BY processing
vi. do algorithmic programming

And SQL is the language I consider using first when I need to:

i. remove duplicates
ii. remerge summary statistics
iii. build a macro variable loaded with a unique list
iv. build a set of macro variables
v. generate a Cartesian product
vi. display a variable with different formats

SQL has been an important part of Base SAS for a number of years now. It provides SAS users with number of alternatives to tradition Data Step and Proc Step processing and a number of unique processing capabilities.

REFERENCES

SAS Institute Inc. (2002-2006), SAS 9.1.3 Language Reference: Dictionary, 5th Edition, Vols 1-4,
Cary, NC: SAS Institute Inc.

SAS Institute Inc. (2008), SAS SQL1: Essentials Course Notes, Book Code E1222
Cary, NC: SAS Institute Inc.

Lafler, Kirk Paul. (2004), PROC SQL: Beyond the Basics Using SAS
Cary, NC: SAS Institute Inc.

CONTACT INFORMATION

Please forward comments and questions to:

Don Boudreaux, Ph.D.
SAS Institute Inc.
11920 Wilson Parke Ave
Austin, TX 78726
Phone: 512.258.5171 Ext. 55265
E-mail: don.boudreaux@sas.com