

PROC SQL - What Does It Offer Traditional SAS® Programming?

Ian Whitlock, Westat Inc.

Abstract

PROC SQL plays two important roles in SAS:

1. To connect SAS with other data base systems
2. To help the typical SAS data processing programmer

This paper looks at the second role. SAS programming problems are presented, where PROC SQL has a distinct advantage over SAS programs that do not use this procedure. By focusing on such problems with coded solutions I hope to answer the question, "When is it most appropriate to consider using PROC SQL in a SAS program?"

Introduction

The typical SAS programmer needs PROC SQL because:

1. It is superb at accessing data stored in multiple data sets at different levels.
2. It can easily produce a Cartesian product.
3. It can perform matching where the condition of a match is not equality.
4. It is good at summarization.
5. With the introduction of 6.11, it can make arrays of macro variables or do away with the need for these arrays by assigning a whole column of values to one macro variable.
6. Macro - SQL interaction enhances both macro and SQL.

In addition to the direct values listed above, one should not underestimate the value of SQL training in teaching one data organization. An example of how SQL can teach data organization is given at the end of the summarizing section.

Matching multiple data sets at different levels

PROC SQL provides a powerful tool when extracting data from different data sets at several different levels. It not only provides simpler code, it provides a new way of looking at these problems.

Suppose we have data at the state, county and city level stored in three data sets.

```
State (state, region,...)
county (cntyid, state, cnty, area, ...)
city (city, cntyid, area, pop,...)
```

Prepare a report of all cities in the midwest with populations over 100,000 with the ratio of the city area to the enclosing county area.

A SAS procedural solution demands that we decide whether to start with states or cities, specify all sorts needed for the various DATA step merges, and specify those merges in detail ending up with a PROC PRINT.

In contrast, SQL asks the fundamental questions:

1. What are the data sets?
2. What are the subsetting conditions?
3. What are the linking conditions?
4. What columns should appear?

```
proc sql ;
  select st.state ,
         cn.cnty ,
         ct.city ,
         ct.area / cn.area as arearato
  from state as st ,
        county as cn ,
        city as ct
  where ct.pop > 100000 and
        st.region = 'MW' and
        st.state = cn.state and
        cn.cntyid = ct.cntyid
  ;
quit ;
```

In all the remaining example code the PROC statement and the QUIT will be omitted.

Cartesian Product

Cartesian product matches are far more common than one-to-one matches, but the MERGE statement assumes one-to-one within BY-groups. To find a Cartesian product match, let's look at a codebook example. I have three data sets:

```
specs ( variable , format )
freq ( variable, format, value, count )
fmts ( format, value, label )
```

The report might look something like this.

first variable using fmtname format

value	label	count
1	first	500
2	sec	0
3	rem	300

second variable using fmt2name format

etc.

Before tackling the problem let's look at the code for joining SPECS and FMTS. The problem involves a Cartesian product because a format may be associated with more than one variable in SPECS, and formats typically have more than one value. Thus FORMAT does not determine a single record in either data set; hence a merge by FORMAT will not work. Note that accomplishing this sort of combining records in a DATA step involves using sophisticated SAS techniques when SQL is not used.

```
create view sfm as
select s.variable ,
       s.format ,
       fm.value ,
       fm.label
from specs as s , fmts as fm
where s.format = fm.format
;
```

It is easy to see how to produce the report with a DATA_NULL_ step when the right information is in a

data set (VARIABLE, FORMAT, VALUE, LABEL, and COUNT). Here is the SQL code to produce the file.

```
create table report as
select
  coalesce (sfm.variable, fq.variable)
    as variable ,
  coalesce ( sfm.format , fq.format )
    as format ,
  coalesce ( sfm.value , fq.value )
    as value ,
  sfm.label ,
  coalesce ( fq.count , 0 ) as count

from sfm full join freq as fq
on sfm.variable = fq.variable and
   sfm.format = fq.format and
   sfm.value = fq.value
order by variable , format , value
;
```

Note that in two SQL statements we have done a lot of the work toward creating a codebook. If one could produce SPECS, FREQ, and FMTS easily, then one could produce a codebook for any properly formatted SAS data set. The FMTS file is trivially produced with the FMTLIB option of PROC FORMAT. The FREQ file requires some macro code. We will postpone discussion of the SPECS file to a later section.

Fuzzy Matching

Fuzzy matching comes in two varieties. In date (or time) line matches one file holds a specific date (or time) and one wants the corresponding record which holds a range of dates (or time). For SAS dates DATE, BEGDATE, and ENDDATE the WHERE clause might be

```
where date is
  between begdate and enddate
```

For efficiency reasons it is important to add an equi-condition whenever possible. In date (or time) matches one often has an ID that must also match, hence the equi-join condition becomes

```
where a.id = b.id and
      date is between begdate and enddate
```

In the other kind of fuzzy matching one cannot trust the identifying variables. Suppose we want to match on social security numbers, SSN, but expect transposition errors and single digit mutations. Now the WHERE clause might be

```
where sum ( substr(a.SSN,1,1) =
            substr(b.SSN,1,1) ,
            substr(a.SSN,2,1) =
            substr(b.SSN,2,1) ,
            ....
            substr(a.SSN,9,1) =
            substr(b.SSN,9,1)
          ) >= 7
```

To make this an equi-join we might add

```
and substr(a.zip,1,3) =
      substr(b.zip,1,3)
```

or some other relatively safe blocking variable.

Summarizing

One PROC SQL step can do the job of a PROC SUMMARY followed by merging of the results with the original data. For example, suppose we have a weighted student sample including many different schools. We want the percentage weight of each student in a school. Then we might have:

```
select school , student , wght ,
       100*wght/sum(wght) as pctwt
from studsamp
group by school
;
```

In this case one gets a message that summary data was remerged with the original data, but that is precisely what we wanted.

Now suppose we want to look at all the students from any school which has some student contributing more than 20% of the weight. The code might be

```
select stu.*
from studsamp as stu ,
```

```
( select distinct school
  from studsamp
  group by school
  having wght/sum(wght) > .2
) as want
where stu.school = want.school
order by stu.school, stu.wght desc
;
```

The technique is important because there are many times one wants to view every one in a group if anyone in the group has some property. SQL provides a natural idiom for producing the report.

Knowing SQL should make one more sensitive to bad patterns of storing data. For example, a common question on SAS-L is how to array data. Given the data

ID	DATE	COUNT
1	5jun1993	50
1	16oct1993	25
1	21dec1993	8
2	14may1990	16
2	27jan1991	3

how do you produce one record per ID with as many date and count fields as needed, say ID, DATE1 - DATE32 and COUNT1 - COUNT32? Another common question is how to work with the arrayed data. For example, how can you compute the rate of decrease in count per month and per year for each ID. The answer is a trivial SQL problem, when the data are stored as they were originally given.

```
select id ,
       (max(count)-min(count))/
       intck('month',min(date),max(date))
       as decpmon,
       calculated decpmon * 12 as decpyr
from origdata
group by id
;
```

After arraying it becomes a harder problem. Perhaps if SAS programmers learned SQL, and how to solve problems without arrays, then they would also learn the advantages of storing data in a non-arrayed form. With SQL training, one comes to realize the importance of putting the information into the data instead of the variable names. Of course, this also

means that the usefulness of SQL is highly dependent on how well the data are stored, but it would be wrong to conclude that one might as well avoid learning SQL because of bad data management practices.

Macro Lists Via PROC SQL

PROC SQL's ability to assign a whole column of values to a macro variable has drastically changed how one writes macro code. Consider the splitting problem. Given a data set ALL with a variable SPLIT naming a member, split ALL by the variable SPLIT. Before version 6.11 one had to use CALL SYMPUT to create an array of data set names and values and then write a monster SELECT statement. The whole thing had to be in a macro in order to repetitively process the array. Now one might view it as a problem to produce two lists

1. The names of data sets
2. WHEN / OUTPUT statements for a SELECT block

The first case is easily handled by

```
select distinct 'lib.'||split
  into :datalist separated by ' '
 from all ;
```

The second is more of the same, only harder.

```
select distinct
  'when ('||split||') output lib.'||
  split into :whenlist
  separated by ';'
 from all
;
```

Now the code to produce the split is trivial and need not even be housed in a macro.

```
data &datalist ;
  set all ;
  select ( split ) ;
  &whenlist ;
  otherwise ;
end ;
run ;
```

In the section on the Cartesian product, we postponed discussion of the data set SPECS. It could be generated from one of the "dictionary" files documented in the Technical Report P-222. Suppose we are interested in making a codebook for the data set LIB.MYDATA, then the following code could generate SPECS.

```
create specs as
select name as variable ,
      case
        when format='' and type='char'
          then $char
        when format='' and type= 'num'
          then best
        else format
      end as format
 from dictionary.columns
 where libname = 'LIB' and
       memname = 'MYDATA'
;
```

To prepare for doing the frequencies needed to make the data set FREQ we could use the array form of generating variables from a column.

```
select variable ,
      format ,
      into :var1 - var9999 ,
      :fmt1 - fmt9999
 from specs
;
```

%let nvar = %sqlobs ;

The frequency data sets can then be generated in a PROC FORMAT with the macro code

```
proc freq data = lib.mydata ;
  %do i = 1 %to &nvar ;
    table &&var&i / out=&&var&i ;
    format &&var&i &&fmt&i... ;
  %end ;
run ;
```

We still have not combined the frequency data sets into one data set, but that task can be left to a competent macro programmer, even one who doesn't know SQL (assuming that that is not a contradiction in terms).

Macro - SQL Interaction

The previous section showed how the making of lists has had a dramatic effect on the way one codes macro problems involving lists. Now we consider a more complex interaction between PROC SQL and macro, where macro code is used to write the SQL code in a loop and the whole problem is much easier, precisely because it is SQL code.

Suppose we have a data set, W, containing the variables NAME and GROUP.

NAME	GROUP
A	1
B	1
B	2
C	2
D	2
D	3
E	3
F	4
G	4
G	5
H	5

We want to collapse groups to the lowest level. For example, since A and B belong to group 1, and B and C belong to group 2, then all members of group 2 are part of group 1 because the groups have the common member B. Once this is seen one can add group 3 to the new group 1 because of the common member D. Thus group 1 covers A, B, C, D and E. Similarly F, G, and H ultimately belong to group 4. More formally, two groups are in the same chain if there is a sequence of groups containing the given groups such that each consecutive pair of groups contains a common name. Using this definition the data set consists of disjoint chains. The problem is to write a program identifying each chain by the minimum group number in the chain.

The intuitive argument given in the previous paragraph uses two kinds of minimization.

1. Find the minimum group (call it MINGROUP) for all names having the same value (e.g. NAME = 'B' has MINGROUP = 1).

2. Find the minimum of all MINGROUP values for all names in a common group (e.g. GROUP = 2 has MINGROUP = 1).

PROC SQL is very suitable to both types of minimization. In the first case we might have

```
create table t as
select name, group,
       min (group) as mingroup
from dataset
group by name ;
```

In the second case we might have

```
create table t as
select name, group,
       min (mingroup) as mingroup
from t
group by group ;
```

These two operations must be repeated over and over until no new minimums are found, since each new extension of a group may mean further collapsing. To express the iteration of this code to an arbitrary level, we need a macro %DO-loop. This time we will present the complete macro, %GROUPIT.

For generality, we make parameters to name the input and output data sets, and the variables represented by NAME, GROUP, and MINGROUP. The parameter MAX is added to insure that the macro does not execute for an excessively long time. (Since the algorithm does converge one could do away with this parameter or set it to the number of observations.)

```
%macro groupit
( data=&syslast, /* input data */
  out=_DATA_, /* resulting data */
  name=name, /* name variable */
  group=group, /* group variable */
  mingroup=mingroup, /* minimum */
  max=20 /* limit #iterations */
) ;

/* -----
   minimize group on name and then group
   repeat until max iterations or done
   ----- */
```

```

%local i done ;

proc sql ;
  /* -----
  initial set up - get first
  minimums, start numbered
  sequence of data sets
  ----- */

  create table __t0 as
  select &name , &group ,
         min (&group) as &mingroup
  from &data
  group by &name ;

  create table __t0 as
  select &name , &group ,
         min (&mingroup) as &mingroup
  from __t0
  group by &group ;

  /* -----
  iterate until done or too many
  iterations
  ----- */

  %do %until (&done or &i > &max) ;
    %let i = %eval ( &i + 1 ) ;

    create table __t&i as
    select &name , &group ,
           min (&mingroup) as &mingroup
    from __t%eval(&i-1)
    group by &name
    ;

    create table __t&i as
    select &name , &group ,
           min (&mingroup) as &mingroup
    from __t&i
    group by &group
    ;

  /* are we finished? */
  reset noprint ;
  select w1.&name
  from __t%eval(&i-1) as w1,
       __t&i as w2
  where w1.&name=w2.&name and
        &group=w2.&group and
        w1.&mingroup ^=
        w2.&mingroup
        ;

  %let done =
    %eval ( not &sqlobs ) ;
  reset print ;
  drop table __t%eval(&i-1) ;
  %end ; /* end iterative loop */

  %if not &done %then
    %put WARNING(GROUPIT):Process
      stopped by condition MAX=&max;
  %else
    %do ;
      create table &out as
      select &name , &group ,
             &mingroup
      from __t&i
      order by &name , &group
      ;
      drop table __t&i ;
    %end ;

  quit ;
  %mend groupit ;

  %groupit ( data = w , name = name1 ,
            group = group1 , out = w2 )

  proc print data = w2 ;
  run ;

```

Conclusion

I have pointed out six areas where SQL code excels. My conclusion is that a good SAS programmer can no longer ignore PROC SQL and remain good.

The author can be contacted by mail at:

Westat Inc.
1650 Research Boulevard
Rockville, MD 20850-3129

or by e-mail at:

whitfo1@westat.com

SAS is a registered trademark or trademark of SAS Institute Inc. in the USA and other countries. ® indicates USA registration