Paper PH06-2012

Get SAS®sy with PROC SQL

Amie Bissonett, Pharmanet/i3, Minneapolis, MN

ABSTRACT

As a data analyst for genetic clinical research, I was often working with familial data connecting parents with their children and looking for trends. When I realized the complexity of this programming could be accomplished quite easily with the SQL Procedure, I started learning the syntax and I've been a PROC SQL fan ever since. I've been suggesting SQL solutions to colleagues for their programming tasks for years and now I'm writing a paper to share the power of PROC SQL with you. This paper will give you some simple PROC SQL steps to get you started and some more complex steps to show you a few of the ways that PROC SQL can make your programs simpler and more efficient.

INTRODUCTION

PROC SQL, which is included with Base SAS, gives you the power of SQL and gives you the use of SAS DATA step functions and macro variables. While the SAS DATA step is a powerful and necessary tool, PROC SQL can provide a much more efficient resolution to many real world programming tasks. This paper provides several examples to demonstrate the many uses of PROC SQL.

PROC SQL SYNTAX

```
PROC SQL ;
  SELECT <DISTINCT> object-item <, ...object-item>
      <INTO macro-variable-specification
        <, ... macro-variable-specification>>
    FROM from-list
    <WHERE sql-expression>
    <GROUP BY group-by-item <, ... group-by-item>>
    <HAVING sql-expression>
    <ORDER BY order-by-item <, ... order-by-item>>
  ;
QUIT ;
```

BASIC MERGING WITH THE INNER JOIN

An INNER JOIN will give you the same results as doing a DATA step merge with data set options of (in=a) and (in=b), and the statement, if a and b ;.

There may be cases when you have two data sets that need to be merged but the variables on which you need to perform the merge have different names. You could rename them, utilize PROC SORT and the DATA step, or you can do it easily in the SQL procedure.

DATA step method:

```
PROC SORT data=san ;
  by subjid ;
run ;
PROC SORT data=francisco out=mergfran (rename=(_pt_=subjid)) ;
  by _pt_ ;
run ;
DATA sfbay ;
  merge san mergfran ;
  by subjid ;
run ;
```

PROC SQL method using an INNER JOIN (the comma between the two data sets is equivalent to 'INNER JOIN'):

```
PROC SQL ;
  create table sfbay as
    select *
      from san a, Francisco b
      where a.subjid=b._pt_
  ;
quit ;
```

The a and b after the input data set names are table aliases. A table alias is a temporary, alternate name for a data set, which allows for a shorter table reference in the PROC SQL statements. See the 'WHERE' statement in this example.

MERGING SELECTING SPECIFIC VARIABLES AND ORDERING THE OUTPUT DATA SET

There may be cases where a data merge is required but the data sets have variables in common. When doing an inner join and using the 'SELECT *' statement, as in the above example, you will get a warning if the same variables exist in both data sets.

```
WARNING: Variable food already exists on file WORK.SFBAY.
WARNING: Variable sites already exists on file WORK.SFBAY.
```

The output dataset will contain the variable with values from the first data set listed in the join. To prevent getting the WARNING the SELECT statement is used where you can specify which data set will provide the variable values. In the following example, subjid is only selected from transfu_yn, or the a. data set. You can also use the ORDER statement to sort your output data set for later processing.

```
PROC SQL ;
  create table transfu as
    select a.*, b.visit, b.eventdt
      from transfu_yn a, transfu_all b
      where a.subjid=b.subjid
      order by a.subjid, b.eventdt
  ;
quit ;
```

subjid	transfuyn
1	Y
3	N
2	N
4	Y

subjid	visit	eventdt	typecd	volume	indiccd
1	W1	3-Jan-11	1	200	1
1	W2	10-Jan-11	1	350	1
4	W7	15-Jul-11	2	590	2
4	W4	23-Jun-11	2	590	2

Table 1. Input data sets, TRANSFU_YN and TRANSFU_ALL

subjid	transfuyn	visit	eventdt
1	Y	W1	3-Jan-11
1	Y	W2	10-Jan-11
4	Y	W4	23-Jun-11
4	Y	W7	15-Jul-11

Table 2. Output data set from CREATE TABLE Statement, TRANSFU

RANGE MERGE AND DISTINCT RECORDS

PROC SQL allows you to merge data based on a range of values, which is not possible in a DATA step merge. In this example, adverse events that occurred between the first dose and last dose of treatment are selected based on the adverse event start date.

```
PROC SQL ;
  create table trt_aes as
    select a.*
      from ds d, ae a
      where d.subjid = a.subjid
        and d.fdosdt <= a.aestdt <= d.ldosdt
    ;
quit ;
```

A range merge is also useful if you want events that occur within a certain amount of time of one another. This example pulls lab records from two different data sets where the lab tests occurred within 28 days of each other.

```
PROC SQL ;
  create table hgb_wbc as
    select distinct a.*, b.b_wbc, b.b_lbdt
    from a_lb a, b_lb b
    where a.subjid=b.subjid and abs(a.a_lbdt - b.b_lbdt) <= 28
  ;
quit;
```

MERGE A DATA SET WITH ITSELF

A common laboratory data summary looks at baseline values against end of study values. PROC SQL can create a data set with these values on one record for each subject in your data.

```
PROC SQL ;
  create table labsum as
    select distinct a.subjid, a.lbtestcd, a.lbstresn as baseline,
      b.lbstresn as eos, a.lbstnrlo, a.lbstnrhi
    from lb a, lb b
    where a.subjid=b.subjid and a.lbtestcd=b.lbtestcd
      and a.visitcd eq 'SCR' and b.visitcd eq 'EOS'
  ;
quit ;
```

subjid	lbtestcd	baseline	eos	lbstnrlo	lbstnrhi
1	WBC	1.79	9.82	4	10.5
2	WBC	6.09	1.81	4	10.5
3	WBC	7.95	11.44	4	10.5
4	WBC	4.73	1.48	4	10.5

Table 3. Output data set from CREATE TABLE Statement, LABSUM.

CARTESIAN PRODUCTS

Note! Cartesian products! Sometimes you actually want them, but be very careful to get all of the merge-by variables, in your WHERE statement, that you'll need to get a clean merge.

When would you want a Cartesian product? PROC SQL is an easy way to create a shell data set for your tables. This example creates a shell data set for a subject disposition table by study cycles.

```
PROC FORMAT cntlout=shell1 (where=(fmtname eq 'SUBJDISP')
                            keep=fmtname start) ;
  value subjdisp
    1 = '  Subjects who started this cycle'
    2 = '  Subjects who discontinued this cycle'
    3 = '  Subjects who completed study this cycle'
    4 = '  Subjects who are continuing beyond this cycle'
  ;
run ;
```

FMTNAME	START
SUBJDISP	1
SUBJDISP	2
SUBJDISP	3
SUBJDISP	4

Table 4. Output data set, SHELL1

In the data set stats, we note that there are no subjects that discontinued in cycle 2 (disp=2) and no subjects completed cycle 3 (disp=3).

cyclecd	disp
1	1
1	2
1	3
1	4
2	1
2	3
2	4
3	1
3	2
3	4

Table 5. Input data set, STATS

```
PROC SQL ;
  create table shell as
    select *
      from (
        select distinct cyclecd
          from stats
        ),
        (select distinct input(trim(left(start)), best.) as disp
          from shell1
        )
    order by cyclecd, disp
  ;
quit ;
```

Obs	cyclecd	disp
1	1	1
2	1	2
3	1	3
4	1	4
5	2	1
6	2	2
7	2	3
8	2	4
9	3	1
10	3	2
11	3	3
12	3	4

Table 6. Output data set from CREATE TABLE Statement, SHELL. The shell data set has all possible disposition statuses for every cycle.

OTHER TYPES OF JOINS – LEFT, RIGHT, FULL

LEFT, RIGHT and FULL JOINS are also used to merge data. Note that with the WHERE statement is replaced with the ON statement for these types of joins.

FULL JOIN, COALESCE FUNCTION

A FULL JOIN is used when all records from both data sets are to be included in the output data set, similar to a DATA step merge. The COALESCE function returns the first non-missing value that is found from the list of expressions enclosed in parentheses.

```
PROC SQL ;
  create table transfu as
    select coalesce(a._pt_, b._pt_) as subjid length=10,
      a.transfuyn, b.*
    from transfu_yn a
    FULL OUTER JOIN
    transfu_all b
    on a._pt_=b._pt_
    order by subjid
  ;
quit;
```

subjid	transfuyn	_pt_	visit	eventdt	typecd	volume	indiccd
1	Y	1	W2	10-Jan-01	1	350	1
1	Y	1	W1	3-Jan-11	1	200	1
2	N	.			.	.	.
3	N	.			.	.	.
4	Y	4	W4	23-Jun-01	2	590	2
4	Y	4	W7	15-Jul-01	2	590	2

Table 7. Output data set from CREATE TABLE Statement, TRANSFU.

LEFT (OR RIGHT) JOIN USING DISTINCT, SUB-QUERY, GROUP BY, AND CASE STATEMENTS

LEFT JOIN and RIGHT JOIN are used for merging when all records of one data set are to be retained and only matching records from the second data set. The LEFT JOIN keeps all records from the first data set and matching records from the second data set, and a RIGHT JOIN does the opposite. In DATA step terms, with data set options of (in=a) and (in=b), a LEFT JOIN is equivalent to ‘if a’ and a RIGHT JOIN is equivalent to ‘if b’ and a FULL JOIN is ‘if a or b’.

This example uses the LEFT JOIN and also introduces the DISTINCT clause, the sub-query, the GROUP BY clause and the CASE statement to create a data set with response information.

```
PROC SQL;
  create table subjresp as
    select distinct a.subjid, a. response, b. numwks,
      case
        when (n(b.fstresp, a.fdosdt)=2) then
          (b.fstresp - a.fdosdt + 1)/7
        else .
      end as timeresp
      from platelet a
      LEFT JOIN
      (select distinct subjid, fdosdt, count(*) as numwks,
          min(visitdt) as fstresp format=date9.
       from (
         select distinct subjid, fdosdt, visitdt
           from platelet
           where response eq 'Y'
       )
       group by subjid
      ) b
      on a.subjid=b.subjid
    order by subjid
  ;
quit ;
```

The resulting output data set contains distinct records for each subject in the platelet data set, which is the first data set in the LEFT JOIN, and matching records from the second data set. The second data set of the LEFT JOIN is the output from nested sub-queries. The first, inner, sub-query, A, selects only those records where a subject had a response of 'Y'. The second, outer, sub-query, B, takes all records from the inner sub-query and uses a couple functions to create summary variables. The COUNT(*) function counts all records and the resulting value is saved in the NUMWKS variable. The MIN function will take the minimum VISITDT value and save it to the FSTRESP variable. The GROUP BY statement in this sub-query is very important as it tells the summary functions to summarize for each subject, or subjid. Without the GROUP BY statement, the COUNT(*) would return the total number of records in the entire data set, and the MIN(visitdt) would be the minimum visitdt in the entire data set.

After the closing parenthesis to the outer sub-query, an alias of b is assigned to this data set, which is then used in the SELECT and ON statements.

The SELECT statement is performing a LEFT JOIN on the PLATELET data set, with alias a, to the sub-query data set, b. The resulting data set will include all records from PLATELET and those from the sub-query that match from the ON statement, a.subjid=b.subjid.

Additional processing is done in the SELECT statement to calculate the number of days from first dose to the date of response. This is done with the use of a CASE statement so that any warnings from missing FSTRESP or FDOSDT are not printed to the log.

The output data set may contain multiple records per subjid if the subject had records both with response = 'Y' and without. An additional sub-query can be wrapped around the query to select the response = 'Y' record if it exists, otherwise select the response = 'N' record. This is shown in the query for Table 8.

subjid	response	numwks	timeresp
1		28	1.1
1	N	28	1.1
1	Y	28	1.1
2		104	2.1
2	N	104	2.1
2	Y	104	2.1
3	N	2	4.9
3	Y	2	4.9
4		.	.
4	N	.	.

Table 8. Output data set from CREATE TABLE Statement, SUBJRESP.

We only want one record per subject with their summary variables and the response variable we want to know if the subject ever had a response. To accomplish this, we can simply add an additional sub-query around the above query. Using the GROUP BY on subjid and taking the max value of response will take a 'Y' value over an 'N' or missing value. We then also use the DISTINCT function to get only one record per subject.

```
PROC SQL ;
  create table subjresp2 as
    select distinct subjid, max(a.response) as response, numwks,
        timeresp
      from (
      | select distinct a.subjid, a. response, b. numwks,
        |   case
        |     when (n(b.fstresp, a.fdosdt)=2) then
        |       (b.fstresp - a.fdosdt + 1)/7
        |     else .
        |   end as timeresp
        |   from platelet a
        |   LEFT JOIN
        |   (select distinct subjid, fdosdt, count(*) as numwks,
        |   |   min(visitdt) as fstresp format=date9.
        |   | from (
        |   | | select distinct subjid, fdosdt, visitdt
        |   B | A   from platelet
        |   | |   where response eq 'Y'
        |   | )
        |   | group by subjid
        |   ) b
      | on a.subjid=b.subjid
      )
    group by subjid
    order by subjid
  ;
quit ;
```

subjid	response	numwks	timeresp
1	Y	28	1.1
2	Y	104	2.1
3	Y	2	4.9
4	N	.	.

Table 9. Output data set from CREATE TABLE Statement, SUBJRESP2.

CREATING MACRO VARIABLES

PROC SQL is very useful for creating macro variables from your data values. You can automate your programs, create loop control variables and easily create table column headers. Multiple queries can be performed within one PROC SQL statement.

PROC SQL automatically produces a PROC PRINT style output when you are not using the CREATE TABLE statement. Adding the NOPRINT option to the PROC SQL statement will prevent this.

SIMPLE MACRO VARIABLES, NOPRINT OPTION

If you have a program that you will use across multiple studies with different number of treatment groups, the following will create macro variables for the number of treatment groups and the number of subjects in each group. The last query is useful if you have a total column and you only want the total for your non-placebo groups.

```
PROC SQL noprint ;
  select trim(left(put(count(distinct cohortcd), best.))) into :ncoh
    from ds
  ;
  select trim(left(put(count(distinct subjid), best.))) into
      :coh1 - :coh&ncoh
    from ds
    group by cohortcd
  ;
  select trim(left(put(count(distinct subjid), best.))) into :coh99
    from ds
    where cohortcd > 1
  ;
quit ;
```

MACRO VARIABLES FOR FORMATTED COLUMN HEADERS

Many summary tables display the treatment group and the number of subjects in the group as the column header. This example shows a simple solution for that need. The MAXTRT macro variable is used in the second query to automate the creation of the column headers. This will work whether there are two treatment groups or ten. It is essential that you use the trim and left options for the MAXTRT macro variable for use in the INTO clause in the second query. By default, the MAXTRT macro variable value will be right justified so the value of 'coh&maxtrt.' will be 'coh 1', which is not a valid SAS name.

```
PROC SQL noprint ;
  select count(distinct trtcd),
      trim(left(put(max(trtcd), best.))) into :ntrt, :maxtrt
    from ds
  ;
  select distinct trim(left(trt)) || '$(N=' ||
      trim(left(put(count(*), best.))) || ')' into :coh1 - :coh&maxtrt.
    from ds
    group by trt
  ;
quit ;
```

The &coh1 macro variable has a value of 'Placebo Cohort-1$(N=13)'. These macro variables can then be used in a PROC REPORT, using '$' as the split character, to display the column header.

DATA CHECKS

A sub-query can be used to easily subset one data set with values from another. In this example, the resulting data set will contain all records from the ALLDATA data set where the subjid is in the CKSUBJ data set. This SQL method is preferred over using DATA steps in case there are multiple records per subject in the CKSUBJ data set.

```
PROC SQL ;
  create table check as
    select *
      from alldata
      where subjid in (
        select distinct subjid
          from cksubj
      )
  ;
quit ;
```

ANALYZING YOUR DATA

PROC SQL may be used to analyze data by calculating simple statistics and formatting the results for output in one easy step.

REPLACE PROC FREQ

PROC FREQ cannot handle a large number of variables in a cross tabulation in the table statement resulting in this error being printed to the log, 'ERROR: The requested table is too large to process.'. PROC SQL to the rescue! This example creates a frequency table with the counts of each distinct combination. All variables in the SELECT statement other than the count variable must also be in the GROUP BY clause.

This example gives a summary of laboratory information.

```
PROC SQL ;
  select distinct trtcd, visitcd, lbtestcd, lbtest, lbstunit, lbstnrlo,
      lbstnrhi, count(*) as n
    from lbcrt
    group by trtcd, visitcd, lbtestcd, lbtest, lbstunit, lbstnrlo,
      lbstnrhi
  ;
quit ;
```

trtcd	visitcd	lbtestcd	lbtest	lbstunit	lbstnrlo	lbstnrhi	n
1	EOS	RBC	Red Blood Cells	10^12/L	4.1	5.6	15
1	EOS	WBC	White Blood Cells	10^9/L	4	10.5	23

Table 10. Output from SELECT statement.

CALCULATE SIMPLE STATISTICS

This example calculates some statistics on dosing data for an analysis data set. The GROUP BY statement is by subjid and cycle so the statistics will be calculated for each unique combination of subjid and cycle.

```
PROC SQL ;
  create table dosevars as
    select distinct subjid, cycle, count(*) as numdoses,
        sum(dose)/count(*) as avgdose, max(dose) as maxdose
      from ex
      where dose > 0
      group by subjid, cycle
      order by subjid, cycle
  ;
quit ;
```

subjid	cycle	numdoses	avgdose	maxdose
1	1	10	675	1000
1	2	4	1000	1000
2	1	8	750	750
2	2	10	750	750
2	3	10	750	750
2	4	8	750	750

Table 11. Output data set from CREATE TABLE Statement, DOSEVARS.

CALCULATE STATISTICS AND FORMAT OUTPUT FOR DISPLAY

This example calculates some statistics on response data and formats the results ready for table display.

```
PROC SQL ;
  create table resprates as
    select distinct trt, bign,
      trim(left(put(count(distinct subjid), best.))) || ' (' ||
      trim(left(put(count(distinct subjid)/bign*100, 7.1))) || ')'
        as nresp
      from (
        select subjid, trt, response, count(distinct subjid) as bign
          from allsubj
          group by trt
      )
      where response eq 'Y'
      group by trt
  ;
quit ;
```

trt	bign	nresp
1	28	23 (82.1)
2	7	6 (85.7)
3	5	5 (100.0)

Table 12. Output from CREATE TABLE Statement, RESPRATES.

CONCLUSION

Whether you're a clinical or statistical programmer, PROC SQL is a great tool for manipulating your data. You can easily create queries to find data issues, subset data sets to further research your data, and merge together multiple data sets, even without same variable names. Simple statistics can be calculated and formatted for output all in one easy step. In addition, while not included in this paper, PROC SQL can also be used to access relational databases directly from SAS if you have a SAS/ACCESS license.

RECOMMENDED READING

Base SAS 9.2 Procedures Guide, PROC SQL

Lafler , Kirk Paul, *Proc SQL: Beyond the Basics Using SAS*®. SAS Institute, 2004.

CONTACT INFORMATION

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

Name: Amie Bissonett
Enterprise: Pharmanet/i3
E-mail: abissonett@pharmanet-i3.com