PharmaSUG 2015 - Paper TT03

PROC SQL: Make it a monster using these powerful functions and options

Arun Raj Vidhyadharan, inVentiv Health, Somerset, NJ
Sunil Mohan Jairath, inVentiv Health, Somerset, NJ

ABSTRACT

PROC SQL is indeed a powerful tool in SAS. However, we can make it even more powerful by using certain PROC SQL functions and options. This paper explores such functions and options that take PROC SQL to the next level.

INTRODUCTION

PROC SQL is often used in our day to day programming for its versatility and efficiency. An efficiently written PROC SQL can get us results much faster than a data step. It is also used for highly complex queries without the overhead of merging multiple datasets using data merge. However, there are certain rarely used functions and options that can be used with PROC SQL, which provides extra flexibility and coding power. Some of these functions are even undocumented. The purpose of this paper is to introduce these functions and options to PROC SQL users so that they can utilize the full potential of SQL coding.

The MONOTONIC function

The automatic variable _N_ in DATA step processing counts the number of times the DATA step begins to iterate. It's very useful when you need the iteration number from the DATA step. Since PROC SQL uses a relational database concept that is different from the DATA step, we can't get the iteration number from the PROC SQL procedure. The MONOTONIC function in PROC SQL can generate very similar result as the _N_ in DATA step. Look at the following example:

Consider the following dataset sashelp.class:

Figure 1.

Name	Sex	Age	Height	Weight
Alfred	M	14	69	112.5
Alice	F	13	56.5	84
Barbara	F	13	65.3	98
Carol	F	14	62.8	102.5
Henry	M	14	63.5	102.5
James	M	12	57.3	83
Jane	F	12	59.8	84.5
Janet	F	15	62.5	112.5
Jeffrey	M	13	62.5	84
John	M	12	59	99.5
Joyce	F	11	51.3	50.5

The following PROC SQL will yield the result as shown below:

Example 1:

```
proc sql;
select monotonic() as rowno, *
from sashelp.class
where monotonic() le 5;
quit;
```

The above program will generate the output:

```
rowno  Name      Sex       Age    Height    Weight
--------------------------------------------------
    1  Alfred    M          14        69     112.5
    2  Alice     F          13      56.5        84
    3  Barbara   F          13      65.3        98
    4  Carol     F          14      62.8     102.5
    5  Henry     M          14      63.5     102.5
```

The MONOTONIC function is quite similar to the internal variable _N_ in DATA Step. We can use it to select the records according to their row number. For example, the following PROC SQL code chooses the observations from the 4th observation to the 8th observation in the sashelp.class dataset.

Example 2:

```
proc sql;
select * from sashelp.class where monotonic() between 4 and 8;
quit;
```

The COALESCE and COALESCEC functions

These functions return the first non-missing value from a list of arguments. COALESCE accepts one or more numeric arguments. The COALESCE function checks the value of each argument in the order in which they are listed and returns the first non-missing value. If only one value is listed, then the COALESCE function returns the value of that argument. If all the values of all arguments are missing, then the COALESCE function returns a missing value. The COALESCE function searches numeric arguments, whereas the COALESCEC function searches character arguments.

Consider the following dataset PATVISIT:

Figure 2.

Patient	Visit1_Date	Visit2_Date	Visit3_Date	Visit4_Date	Visit5_Date
100		8-Feb-14		15-Apr-14	21-May-14
101				12-Apr-14	
102			9-Mar-14	19-Apr-14	
103	4-Jan-14		16-Mar-14		
104					
105			2-Mar-14		12-May-14

If we want to find the first non-missing visit date for each patient, we could use the COALESCE function in PROC SQL as follows:

Example 3:

```
proc sql;
   create table patvisit2 as select *, coalesce(visit1_date, visit2_date,
visit3_date, visit4_date, visit5_date) as first_visit_date from patvisit;
quit;
```

The resulting dataset patvisit2 would look like this:

Figure 3.

Patient	Visit1_Date	Visit2_Date	Visit3_Date	Visit4_Date	Visit5_Date	first_visit_date
100		8-Feb-14		15-Apr-14	21-May-14	8-Feb-14
101				12-Apr-14		12-Apr-14
102			9-Mar-14	19-Apr-14		9-Mar-14
103	4-Jan-14		16-Mar-14			4-Jan-14
104						
105			2-Mar-14		12-May-14	2-Mar-14

The SPEDIS and SOUNDEX functions

The SPEDIS function (stands for spelling distance) is used for fuzzy matching, which is comparing character values that may be spelled differently. The logic is a bit complicated, but using this function is quite easy. As an example, suppose you want to search a list of names to see if the name Friedman is in the list. You want to look for an exact match or names that are similar. Here is such a program:

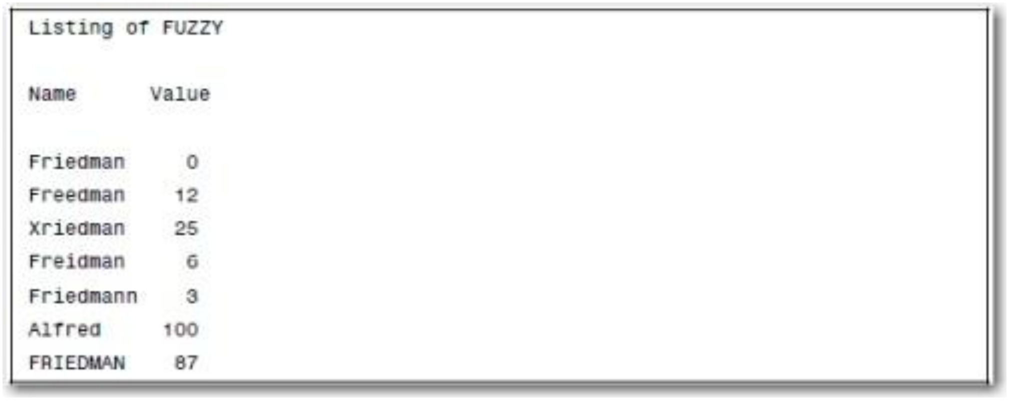
Using the SPEDIS function to perform a fuzzy match:

Example 4:

```
data fuzzy;
input Name $20.;
Value = spedis(Name,'Friedman');
datalines;
Friedman
Freedman
Xriedman
Freidman
Friedmann
Alfred
FRIEDMAN
;
run;
```

Here is a listing of dataset Fuzzy:

Figure 4.



```
Listing of FUZZY


Name       Value


Friedman      0
Freedman     12
Xriedman     25
Freidman      6
Friedmann     3
Alfred      100
FRIEDMAN     87
```

The SPEDIS function returns a 0 if the two arguments match exactly. The function assigns penalty points for each type of spelling error. For example, getting the first letter wrong is assigned more points than misspelling other letters. Interchanging two letters is a relatively small error, as is adding an extra letter to a word.

Once the total number of penalty points has been computed, the resulting value is computed as a percentage of the length of the first argument. This makes sense because getting one letter wrong in a 3-letter word would be a more serious error than getting one letter wrong in a 10-letter word.

Notice that the two character values evaluated by the SPEDIS function are case-sensitive (look at the last observation in the listing). If case may be a problem, use the UPCASE or LOWCASE function before testing the value with SPEDIS.

To identify any name that is similar to **Friedman**, you could extract all names where the value returned by the SPEDIS function is less than some predetermined value. In the program here, values less than 15 or 20 would identify some reasonable misspellings of the name.

For human names, we can check similarities by the SOUNDEX function to avoid duplicates. The SASHELP.CLASS has 11 names. Phonically, John and Jane look similar according to the SOUNDEX function. Note that the SOUNDEX algorithm is English-biased and is less useful for languages other than English.

Example 5:

```
proc sql;
select a.name as name1, b.name as name2
from sashelp.class as a, sashelp.class as b
where soundex(name1) = soundex(name2) and (name1 gt name2);
quit;
```

SAS Output:

```
name1     name2
------------------
John      Jane
```

The IFC and IFN functions

The two functions play a role like the CASE-WHEN-END statements in typical SQL syntax, if the condition is about a binary selection. The IFC function deals with character variables, while the IFN function is for numbers. In the following example, IFN evaluates the expression TotalSales > 10000. If total sales exceed $10,000, then the sales commission is 5% of the total sales. If total sale is less than $10,000, then the sales commission is 2% of the total sales.

Example 6:

```
data Sales;
input TotalSales;
datalines;
25000
10000
500
10300
;
run;

proc sql;
select TotalSales,
ifn(TotalSales ge 10000, TotalSales*.05, TotalSales*.02) as commission from Sales;
quit;
```

The output:

```
TotalSales  commission
----------------------
25000        1250
10000         500
  500          10
10300         515
```

COLON (:) MODIFIER COUNTERPART IN PROC SQL

In the DATA step, the colon operators (such as =:, >:, and <=:) can be used for truncated string comparisons. In PROC SQL, the following truncated string comparison operators are available: EQT, GTT, LTT, GET, LET and NET. These operators compare two strings after making the strings the same length by truncating the longer string to the same length as the shorter string. The truncation is performed internally. Neither operand is permanently changed. The following table lists the truncated comparison operators:

Table 1.

Symbol	Definition	Example
EQT	equal to truncated strings	where Name eqt 'Aust';
GTT	greater than truncated strings	where Name gtt 'Bah';
LTT	less than truncated strings	where Name ltt 'An';
GET	greater than or equal to truncated strings	where Country get 'United A';
LET	less than or equal to truncated strings	where Lastname let 'Smith';
NET	not equal to truncated strings	where Style net 'TWO';

JOIN METHOD CHOSEN BY PROC SQL

If you want to improve the join performance of programs that use PROC SQL to join tables, you need to know how the PROC SQL query optimizer chooses the join methods. There is an undocumented option _METHOD on the PROC SQL statement that will display the hierarchy of processing methods that will be chosen by PROC SQL. You need to set the SAS System option MSGLEVEL = I to see the internal form of the query plan in the SAS LOG. The PROC SQL _METHOD option can be used as an effective way to analyze a query process as well as for debugging purposes. An undocumented option _TREE will show the hierarchy tree as planned in the SAS LOG.

The PROC SQL execution methods include:

sqxcrta	Create table as Select
sqxslct	Select
sqxjsl	Step Loop Join (Cartesian)
sqxjm	Merge Join
sqxjndx	Index Join
sqxjhsh	Hash Join
sqxsort	Sort
sqxsrc	Source Rows from table
sqxfil	Filter Rows
sqxsumg	Summary Statistics (with GROUP BY)
sqxsumn	Summary Statistics (not grouped)
sqxuniq	Distinct rows only

Example 7:

```
OPTIONS MSGLEVEL=I;
proc sql  _METHOD _TREE;
select a.name,a.weight,b.predict
from sashelp.class a, sashelp.classfit b
where a.name=b.name;
quit;
```

The above program will display the query plan below in the SAS LOG file:

```
      sqxslct
          sqxjhsh
              sqxsrc( SASHELP.CLASS(alias = A) )
              sqxsrc( SASHELP.CLASSFIT(alias = B) )


Tree as planned.


                               /-SYM-V-(a.Name:1 flag=0001)
                     /-OBJ----|
                    |         |--SYM-V-(a.Weight:5 flag=0001)
                    |          \-SYM-V-(b.predict:6 flag=0001)
           /-JOIN---|
          |         |                              /-SYM-V-(a.Name:1 flag=0001)
          |         |                    /-OBJ----|
          |         |                   |          \-SYM-V-(a.Weight:5 flag=0001)
          |         |          /-SRC----|
          |         |         |          \-TABL[SASHELP].class opt=''
          |         |--FROM---|
          |         |         |                    /-SYM-V-(b.predict:6 flag=0001)
          |         |         |          /-OBJ----|
          |         |         |         |          \-SYM-V-(b.Name:1 flag=0001)
          |         |          \-SRC----|
          |         |                    \-TABL[SASHELP].classfit opt=''
          |         |--empty-
          |         |          /-SYM-V-(a.Name:1)
          |          \-CEQ----|
          |                    \-SYM-V-(b.Name:1)
 --SSEL---|
```

ALL, ANY, AND SOME

ALL, ANY and SOME are special predicates that are oriented around subqueries. They are used in conjunction with relational operator. Actually, there are only two because ANY and SOME in SQL procedure are the same. ALL, ANY, and SOME are very similar to the IN predicate when it is used with subqueries; they take all the values produced by the subquery and treat them as a unit. However, unlike IN, they can be used only with subqueries. In SQL procedure, ALL and ANY differ from each other in how they react if the subquery procedures have no values to use in a comparison. These differences can give your queries unexpected results if you do not account for them. One significant difference between ALL and ANY is the way they deal with the situation in which the subquery returns no values. In SQL procedure, whenever a legal subquery fails to produce output, the comparison operator modified with ALL is automatically TRUE, and the comparison operator modified with ANY is automatically FALSE.

Example 8:

```
Proc SQL;
select name, age from sashelp.class where age > ALL
(select age from sashelp.class where age >16);

select name, age from sashelp.class where age > ANY
(select age from sashelp.class where age >16);

select name, age from sashelp.class where age > SOME
(select age from sashelp.class where age >16);
quit;
```

The first query would produce the entire list of SASHELP.CLASS values as following results,

```
Name           Age
------------------
Alfred          14
Alice           13
Barbara         13
Carol           14
Henry           14
James           12
Jane            12
Janet           15
Jeffrey         13
John            12
Joyce           11
Judy            14
Louise          12
Mary            15
Philip          16
Robert          12
Ronald          15
Thomas          11
William         15
```

whereas the second and third queries would produce no output.

DOUBLE option for Inserting a Blank Row into Output

SQL can be made to automatically insert a blank row between each row of output. This is generally a handy feature when a single logical row of data spans two or more lines of output. By having SQL insert a blank row between each logical record (observation), you create a visual separation between the rows – making it easier for the person reading the output to read and comprehend the output. The DOUBLE option is specified as part of the SQL procedure statement to insert a blank row and is illustrated in the following SQL code.

Example 9:

```
proc sql DOUBLE;
select * from sashelp.cars;
quit;
```

CONCLUSION

Using the functions and options discussed above can drastically improve your programming approach and overall efficiency. These are added tools to your programming inventory that can be used in appropriate situations and hence avoid multiple data steps.

REFERENCES

Helpful Undocumented Features in SAS®
Wei Cheng, ISIS Pharmaceuticals, Inc., Carlsbad, CA

Top 10 Most Powerful Functions for PROC SQL
Chao Huang. Oklahoma State University
Yu Fu. Oklahoma State University

Handling Missing Values in the SQL Procedure
Danbo Yi, Abt Associates Inc., Cambridge, MA
Lei Zhang, Domain Solutions Corp., Cambridge, MA

A Hands-On Tour Inside the World of PROC SQL
Kirk Paul Lafler, Software Intelligence Corporation

ACKNOWLEDGMENTS

The authors would like to thank John Durski, Associate Director, inVentiv Health, for all his support and motivation in writing this paper.

CONTACT INFORMATION

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

Name: Arun Raj Vidhyadharan
Enterprise: inVentiv Health
Address: 500 Atrium Drive
City, State ZIP: Somerset, NJ 08873
Work Phone: 732.652.3490
E-mail: arunraj.vidhyadharan@inventivhealth.com

Name: Sunil Mohan Jairath
Enterprise: inVentiv Health
Address: 500 Atrium Drive
City, State ZIP: Somerset, NJ 08873
Work Phone: 732.652.3482
E-mail: sunil.jairath@inventivhealth.com