PhUSE 2012

Paper IS03

The Power of PROC SQL

Amanda Reilly, ICON, Dublin, Ireland

ABSTRACT

PROC SQL can be a powerful tool for SAS® programmers. This paper covers how using PROC SQL can improve the efficiency of a program. It also aims to make programmers more comfortable with using PROC SQL by detailing the syntax and giving a better understanding of the differences between the SQL procedure and the DATA step.

The focus of this paper will be to detail the advantages of using PROC SQL and to help programmers identify when PROC SQL is a preferred choice in terms of performance and efficiency. This understanding will enable programmers to make an informed decision when choosing the best technique for processing data, without the discomfort of being unfamiliar with the PROC SQL option.

This paper is targeted at an audience who are relatively new to programming within the pharmaceutical industry.

INTRODUCTION

PROC SQL is one of the built-in procedures in SAS, and it can be a powerful tool for SAS programmers as it can sort, subset, merge, concatenate datasets, create new variables and tables, and summarize data all at once. Programmers can improve the efficiency of a program by using the optimization techniques used by PROC SQL.

Syntax

Understanding the syntax of PROC SQL, as well as the differences between PROC SQL and base SAS, is imperative to programmers using PROC SQL correctly, and to understanding other programmers' code which may need to be used or updated.

```
proc sql <OPTIONS>;
create table <NewDataset> AS
   select <distinct><column names>
    from <OldDataset>
    <where>
    <group BY>
   <order BY>;
quit;
```

Creating a new dataset
Variables in the new dataset
Where to take data from

i.e. Summarizing by treatment group,
Built-in Proc Sort
Similar to a nodupkey

```
proc sql <OPTIONS>;
create table <NewDataset> AS
  select <distinct><column names>
  from <OldDataset>
    <group BY>
    <having>;
quit;
```

The having clause works with the GROUP BY clause to restrict the queries results with a condition

To include all values from OldDataset in NewDataset, a * can be used to represent all variables, so that 'select OldDataset.*' means 'select all variables' from OldDataset. The 'distinct' function is very useful for counting unique values, rather than the total number of values. For example, SUBJID might be repeated over five lines for each subject, but the total number of subjects in a dataset (N) can be found using 'count (distinct SUBJID)'.

The 'group by' statement is another that is particularly useful for programmers in the pharmaceutical industry, as it is often necessary to summarize data over various groups such as treatments, parameters, or

responses. Using the 'by' statement in a DATA step needs the data to already be sorted by those variables, but the 'group by' statement in PROC SQL doesn't require any sorting. The example below was grouped by 'level' to compute the number and percentage of subjects at each level. It shows that almost 44% of the 119 subjects are under 45yrs old, less than 1% are over 65 and just over 55% are in between. A similar procedure might be used to examine the data at a treatment level.

```
/*N*/
proc sql;
create table TOTN as
select VARIABLE, LEVEL, TRT, COUNT, sum (COUNT) as N
    from DEMOG
    group by VARIABLE;
quit;

/*Total n & %*/
proc sql;
create table TOTAL as
select VARIABLE, LEVEL, TRT, COUNT, sum (COUNT) as LevelCOUNT,
((sum(COUNT) / N)* 100/1) as LevelPC format=8.2, N
    from TOTN
    group by LEVEL                          (Option: where TRT ne '')
    order by VARIABLE, LEVEL, TRT;          'Order by' allows us to output a sorted dataset
quit;
```

	Variable	Level	Treatment	Frequency Count	LevelCOUNT	LevelPC	N
1	AGEGRP	<45 years		2	52	43.70	119
2	AGEGRP	<45 years	PLACEBO	14	52	43.70	119
3	AGEGRP	<45 years	TREATMENT A	25	52	43.70	119
4	AGEGRP	<45 years	TREATMENT B	11	52	43.70	119
5	AGEGRP	>65	PLACEBO	0	1	0.84	119
6	AGEGRP	>65	TREATMENT A	1	1	0.84	119
7	AGEGRP	>65	TREATMENT B	0	1	0.84	119
8	AGEGRP	>=45 to <=65 years		3	66	55.46	119
9	AGEGRP	>=45 to <=65 years	PLACEBO	13	66	55.46	119
10	AGEGRP	>=45 to <=65 years	TREATMENT A	34	66	55.46	119
11	AGEGRP	>=45 to <=65 years	TREATMENT B	16	66	55.46	119

Variable	Level	Treatment	Frequency Count	LevelCOUNT	LevelPC	N
RACE	ASIAN	PLACEBO	0	0	0.00	119
RACE	ASIAN	TREATMENT A	0	0	0.00	119
RACE	ASIAN	TREATMENT B	0	0	0.00	119
RACE	BLACK	PLACEBO	0	0	0.00	119
RACE	BLACK	TREATMENT A	0	0	0.00	119
RACE	BLACK	TREATMENT B	0	0	0.00	119
RACE	OTHER	PLACEBO	0	0	0.00	119
RACE	OTHER	TREATMENT A	0	0	0.00	119
RACE	OTHER	TREATMENT B	0	0	0.00	119
RACE	WHITE		5	119	100.00	119
RACE	WHITE	PLACEBO	27	119	100.00	119
RACE	WHITE	TREATMENT A	60	119	100.00	119
RACE	WHITE	TREATMENT B	27	119	100.00	119

The select line lists the variables that the new dataset being created will contain. The variables will appear in the new dataset in the same order that they are listed in this statement, so keeping this in mind will avoid needing to use a retain statement at a later point. The 'select' statement can contain lots of information about the variables, such as the variable attributes, and using it to its full potential will maximize the efficiency of a program.

A function can be performed on an existing variable (var1) and a new variable (var2) can be created to hold the value, using 'function (var1) as var2'. Similarly, the code 'var1 as var2' can be used to rename 'var1' to 'var2'. The following example computes the basic statistics of the data across each treatment.

```
select
   trt                                  format=$20.   label='Treatment',
   count(distinct SUBJID) as N          format=8.     label='Total',
   mean(X) as AVERAGE                   format=8.     label='Average Value',
   std(X) as STDDEV                     format=8.     label='Standard Deviation',
   min(X) as MINIMUM                    format=$20.   label='Minimum Value',
   max(X) as MAXIMUM                    format=$20.   label='Maximum Value',
from olddata
group by trt;
```

WILDCARD

PROC SQL also includes a wildcard option (%) which searches for values starting with, ending with or containing certain strings or characters, by using '%' to replace the unknown part:

- 'x%' returns all values beginning with 'x'
- '%x' returns all values ending with 'x'
- '%x%' returns values containing 'x'

Example:
All reported adverse events for terms including the word 'pain' can be found using the following statement:

```
where aeterm = '%pain%'
```

This is equivalent to '`if index(aeterm,'pain')>0`'

INSERT

Another useful statement is the 'insert' statement, which inserts either values or variables into an existing dataset. This can be done in just one PROC SQL step, whereas in base SAS a dummy dataset would need to be created and then appended to the existing dataset.

Example – values:

```
proc sql;
    insert into NewDataset (var1, var2, var3)
    values (3, 5, 'XXX');
quit;
```

Example – variables:

```
proc sql;
   insert into NewDataset (var1, var2, var3)
   select var4, var5, var6 from OldDataset;
quit;
```

This can be really useful if, for instance, a blank row needs to be inserted between categories.

PROC SQL SYNTAX VS. BASE SAS CODE SYNTAX

There are a number of differences between PROC SQL syntax and base SAS syntax that programmers need to remember when programming. There are three main differences:

1. Variables in a list (for example, in a select statement) in PROC SQL are separated with a comma between each variable. In Base SAS code, they are only separated with a space.

2. A data step or procedure in base SAS is ended with a 'run' statement. A PROC SQL procedure is ended using 'quit'.

3. Every statement in base SAS code ends with a semi-colon, but in PROC SQL only one semi-colon follows the body of the procedure code (which contains the select, from, group and other optional statements).

There are some other differences also such as 'where trtsdt <= adt <= trtedt' in base SAS would be coded as 'where

adt between trtsdt and trtedt'. Code such as 'where subjid=' '' or 'where subjid=.' is used in base SAS to consider missing/null data and whether the variable is character or numeric would need to be considered to do this. However, 'where subjid is missing' can be used for both character and numeric variables in PROC SQL. Similarly, blanks can be inserted into both character and numeric variables using 'insert into Dataset (var1, var2, var3) values (null, null, null)', which will correctly insert a ' ' or '.', depending on whether the variable is a character or numeric variable.

MACRO VARIABLES

Macro variables are another example of optimizing the benefits of PROC SQL. A macro variable can be used to hold a value to be used later in the program. The following simple example would insert the maximum AVISITN value into 'maxvis':

```
Proc sql;
   select max(AVISITN) into: maxvis
   from libnew.adlb;
quit;
```

In base SAS a proc means and a call symput would need to be used. PROC SQL is also useful for doing a number of steps at the same time, as shown in another example below.

```
*** Create macro variables ***;
Proc SQL noprint;

* Number of treatments *;                    Creates variable 'trtcnt' with the number of treatments in SL
   select count(distinct trtan) into: trtcnt
   from SL;

                                   A do loop from 1 to trtcnt which creates trtn1, trtn2,..,trtntrtcnt
     %do i = 1 %to &trtcnt;
     * Treatment names for each macro variable *;
        select distinct trta into: trtn&i
        from SL
        where trtan = &i;

        * Counts for each treatment *;  <------------ Number of subjects in each treatment
        select count(distinct SUBJID) into: trtcnt&i
        from SL
        where trtan = &i;
    %end;

Quit;

****resolving the macro to know the count****;
%do i = 1 %to &trtcnt;
   %put trtn&i:&&trtn&i trtcnt&i:&&trtcnt&i;
%end;
%put trtcnt:&trtcnt;
```

COMBINING DATASETS

To create a dataset using data from two or more separate datasets, the 'merge' statement in a SAS data step can be used, after first sorting all datasets by the same 'by variables' being merged on. Equivalently, a 'join' statement can be used to create one dataset from two or more datasets in PROC SQL and the capabilities of the join statement in PROC SQL is one of the main reasons it may need to be introduced into our programming. To decide whether a 'merge' or a 'join' is more appropriate for our data, the sizes of the datasets, the relationship between them, and the sparseness or density of the matches must be considered.

There is one characteristic of a join that will always be an advantage over a merge; data can be joined without it needing to be sorted first. This eliminates the need for pre-processing the data before merging it, therefore reducing the number of steps involved. Other advantages are that joins can be performed when key variables have different names, and multiple datasets can be joined on different levels, which would take many proc sorts and data steps in base SAS.

INNER AND OUTER JOINS

Inner join (if a and b) – *use WHERE*

```
proc sql;
   create table newdata as
   select column1,…, columnN
   from table1, table2
   where variables;
quit;
```

Select certain variables

Select ALL variables

Example:

```
select *
from dset1 A, dset2 B
where A.subjid=B.subjid and A.visit=B.visit;
```

BASE SQL

```
data newdata;
   merge dset1 dset2;
   by subjid visit;
   if a and b;
```

Omitting the 'where' statement outputs all combinations ('Cartesian product'), which can be very useful for creating dummy datasets (for example, all subjects combined with each visit).



Left Outer ('if a') / Right Outer ('if b') / Full Joins ('if a or b') – *use ON*

```
proc sql;
   create table newdata as
   select column1,…, columnN
   from table1 JOIN TYPE (LEFT JOIN/RIGHT JOIN/FULL JOIN) tableM
   on variables;
quit;
```

Select certain variables

Select ALL variables

Example:

```
select *
from dset1 A LEFT JOIN dset2 B
on A.subjid=B.subjid and A.visit=B.visit;
```

BASE SQL

```
data newdata;
   merge dset1(in=a) dset2(in=b);
   by subjid visit;
   if a;
```

Similarly, a right join equates to a 'if b' merge, and a full join equates to a 'if a or b' merge. PROC SQL can also perform the equivalent of a 'if a not b' / 'if b not a' data step merge by using a left/right join procedure as above, and adding a where clause (b. subjid is null/a.subjid is null).

Example:

```
select *
from dset1 A LEFT JOIN dset2 B
on A.subjid=B.subjid and A.visit=B.visit
where B.subject is null
```

BASE SQL

```
data newdata;
   merge dset1(in=a) dset2(in=b);
   by subjid visit;
   if a and not b;
```

COALESCE FUNCTION

It's important to note that there is a difference between how joins and merges are performed; a join doesn't overlay common variables in the way a merge does. Therefore, if SUBJID is common to both datasets, performing a left join on SUBJID may cause an error in the logs unless SUBJID is renamed in one of the datasets. This is also an issue with full joins, but can be resolved using the 'coalesce' function. Without this, the output can be inaccurate. The 'coalesce' function will pick the first non-missing value, and overlay the variable being joined by.

Example:

There are 7 subjects in the demography dataset and another 7 in the protocol deviations dataset, but there is one subject in each dataset that is not in the other. A full join is needed to see all 8 subjects in the new dataset.

SUBJID	TRT	Race	Sex	Age Group
BU001IM-31-010	Treatment #1	WHITE	F	<45 years
BU001IM-31-011	Placebo	WHITE	F	<45 years
BU001IM-31-012	Treatment #2	WHITE	F	>=45 to <=65 years
BU001IM-31-013	Treatment #2	WHITE	F	<45 years
BU001IM-31-014	Treatment #1	WHITE	F	<45 years
BU001IM-31-015	Treatment #2	WHITE	F	<45 years
BU001IM-31-016	Placebo	WHITE	F	<45 years

SUBJID	TRT	Any Major Protocol deviation?
BU001IM-31-010	Treatment #1	No
BU001IM-31-011	Placebo	No
BU001IM-31-012	Treatment #2	Yes
BU001IM-31-014	Treatment #1	No
BU001IM-31-015	Treatment #2	No
BU001IM-31-016	Placebo	No
BU001IM-31-017	Treatment #1	Yes

When these datasets are joined using the following procedure, the ID of subject 31-017 has been lost since the join didn't overlay the SUBJID variable the way a merge statement would.

```
proc sql;
   create table newdata as
   select a.*, b.*
   from dset1 A FULL JOIN dset2 B
   on A.subjid=B.subjid;
quit;
```

SUBJID	TRT	Race	Sex	Age Group	Any Major Protocol deviation?
BU001IM-31-010	Treatment #1	WHITE	F	<45 years	No
BU001IM-31-011	Placebo	WHITE	F	<45 years	No
BU001IM-31-012	Treatment #2	WHITE	F	>=45 to <=65 years	Yes
BU001IM-31-013	Treatment #2	WHITE	M	<45 years	
BU001IM-31-014	Treatment #1	WHITE	F	<45 years	No
BU001IM-31-015	Treatment #2	WHITE	M	<45 years	No
BU001IM-31-016	Placebo	WHITE	F	<45 years	No
					Yes

To correct this, the 'coalesce' function should be used on the two common variables (TRT and SUBJID):

```
proc sql;
   create table newdata (drop=subjid subjid2 trt trt2) as
   select a.*, b.*,

   coalesce (a.trt, b.trt2) as FULL_TRT,
   coalesce (a.subjid, b.subjid2) as FULL_SUBJID

   from dset1 A FULL JOIN dset2 (rename= (subjid=subjid2 trt=trt2)) B
   on A.subjid=B.subjid2;
quit;
```

This produces a full join correctly.

FULL_SUBJID	FULL_TRT	Race	Sex	Age Group	Any Major Protocol deviation?
BU001IM-31-010	Treatment #1	WHITE	F	<45 years	No
BU001IM-31-011	Placebo	WHITE	F	<45 years	No
BU001IM-31-012	Treatment #2	WHITE	F	>=45 to <=65 years	Yes
BU001IM-31-013	Treatment #2	WHITE	M	<45 years	
BU001IM-31-014	Treatment #1	WHITE	F	<45 years	No
BU001IM-31-015	Treatment #2	WHITE	M	<45 years	No
BU001IM-31-016	Placebo	WHITE	F	<45 years	No
BU001IM-31-017	Treatment #1				Yes

JOINING SUBSETS OF DATASETS

Another useful option in PROC SQL is that, in one step, two datasets can be joined together while including a where statement on one dataset so that only a subset of that data is being joined. This might be particularly useful when comparing baseline data with that data from other visits, as a subset of the data where records have baseline flag 'Y' needs to be compared with all other records.

Example:

```
proc sql;
   create table newdata as
   select a.*, b.base, a.aval - b.base as basechange
   from
   dset1 A
        LEFT JOIN
   (select usubjid, aval as base from dset1 where ablfl='Y') as B
   on a.usubjid=b.usubjid
   order by usubjid, avisitn;
quit;
```

This is also an example of a *self-join* as both A and B are coming from 'dset1' originally.

More options can also be added onto the 'on' statement. For instance, if only interested in comparing data from visits after the baseline visit, the 'on' statement can be updated to '`on a.usubjid=b.usubjid` **`and avisitn >1`**'. In SAS base code a subset would need to be created first, and then both the full dataset and the subset would need to be sorted before merging them in another data step, and then using a proc sort to order them properly by subject and visit.

FUZZY MERGE

A fuzzy merge is another example of something that would take many steps in base SAS code but can be done in one simple PROC SQL step. Unlike normal merging, which involves two or more datasets that have one or more common variables which have exact matching values in each, fuzzy merging is the process of matching records where the condition doesn't apply to an exact match but, perhaps, on a value being in a certain range.

Example:
A programmer needs to know which dose a subject was taking at the time of a particular adverse event. In base SAS code this could be a messy process involving multiple proc sorts and data steps, or it takes just one PROC SQL step!

```
proc sql;
   create table newdata as
   select a.*, b.exdose as AEDOSE from
   ADAE A  LEFT JOIN
   (select usubjid, exstdt, exendt, exdose from ADEX) as B

   on a.usubjid=b.usubjid and aestdt between exstdt and exendt;
quit;
```

MULTIPLE LEVEL MERGE

To merge multiple datasets together on variables which are common to all datasets being merged, each dataset being merged needs to be sorted and then the 'merge' statement in a data step can be used. But what if multiple datasets are being merged on different levels? For example, what if dataset1 and dataset2 are being merged on variables common to them, but dataset3 is also being merged in on variables which are also in dataset2 but not dataset1? This would mean

- Sorting the first two datasets on the variables common to them
- Merging these datasets
- Sorting the merged data and the last dataset
- Merging these datasets

Therefore, there are a lot of steps involved for each level of merging. Again, PROC SQL can do all of this in just one step!

Example:
A programmer needs to join ADaM.ADSL and SDTM.VS on USUBJID, and also needs to merge this with SDTM.SV on VISITNUM. Note that USUBJID is common to all three, but VISITNUM is only common to B and C, not A.

```
proc sql;
   create table newdata as
   select a.*, b.*, c.*
   from

   (select * from ADAM.ADSL) as A,
   (select usubjid, visitnum, vstestcd, vsstresn from SDTM.VS) as B,
   (select usubjid, visitnum, svstdtc from sdtm.sv) as C

   where a.usubjid=b.usubjid and b.usubjid=c.usubjid and c.visitnum=b.visitnum;
quit;
```

CONCLUSION

This paper has demonstrated that PROC SQL is very powerful as a lot of functions and statements can be included in just one step. This one step can replace multiple steps of bulky base SAS code because the need for pre-processing the data is eliminated.

Reducing the volume of code in a program can improve the clarity and legibility of the program and make it much easier to follow, as comments placed above a SQL procedure can detail exactly what the step is doing. This is imperative if programmers need to use, update or debug another programmer's code in the future. In base SAS, a comment may apply to multiple steps which include pre-processing the data before the desired result can be achieved. It is often very difficult to see where the code that the comment applies to ends, as a programmer must follow the train-of-thought of the author to see what each step is doing. Even though the person picking up the code may be very inexperienced in PROC SQL and may not understand the code, if a PROC SQL step is used instead they will at least be able to pinpoint exactly what is happening in each procedure using the comments above, which will make it a lot easier to decide where updates need to be made.

As well as being very useful for computing basic statistical functions (often needed to quickly summarise data for validation purposes), PROC SQL is also very beneficial when combining datasets. Datasets can be joined when the data is not sorted or when key variables have different names in the datasets being joined, and multiple datasets can be joined on variables which are not common to all datasets. It is, therefore, often more efficient to use one PROC SQL step rather than multiple sort procedures and data steps to produce the exact same thing.

On the other hand, there are some drawbacks to using PROC SQL. As well as the fact that other programmers may not be able to understand the author's program, the author's lack of understanding may result in incorrect results. As detailed earlier in the paper, a full join may cause an error if the coalesce function is not used, and all many-to-many joins need careful attention to ensure accuracy. It is, therefore, important for programmers to

be able to identify when PROC SQL is the preferred choice, and when it can be a weaker choice. This will depend on the size of the datasets, the relationship between them, and the sparseness or density of the matches. In general, a merge statement in a data step will be more efficient (in terms of CPU run-time) than a SQL procedure for a simple join, whereas an inner join in a SQL procedure will be more efficient if the datasets are large and unsorted.

Another thing worth considering is the amount of time that it will take for a programmer to use PROC SQL if they are not very comfortable with it. The efficiency that may be gained in run-time or program space may not be worth the time taken to write the program, if a more cumbersome piece of code that would produce the same results may be written far quicker. This may be something that is dependent on how often the code may be used.

In conclusion, PROC SQL can be a very powerful tool for programmers in the pharmaceutical industry as it can improve the efficiency of a program in many ways, and this efficiency will only increase as the programmer's understanding of it improves.

CONTACT INFORMATION

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

Author Name: Amanda Reilly
Company: ICON Clinical Research Ltd
Address: South County Business Park, Leopardstown
City / Postcode: Dublin 18
Email: Amanda.Reilly@iconplc.com
Web: http://www.iconplc.com/