Paper CT06

Innovative SQLs

Joseph Murphy, IQVIA, Reading, United Kingdom

ABSTRACT

Proc SQL is the precocious younger brother in our industry, useful but annoying. Any semi-experienced programmer cannot deny their many applications. Yet it tends to be an unspoken rule, in many CROs and Pharma companies, that they are used as a “last resort” or in very specific circumstances. To a layman, the role of a programmer is often considered as “black and white”. It is finite, exact, left-lobal: no room for creativity or innovation. However, standing from the other side of this gulf, we can confirm this is not the case. There is much room for creativity and abstract thinking when programming and, in a lot of cases, SQL code can be the vehicle for this. This presentation will attempt to drive creative uses for Proc SQL code as well as shed light on the typical uses for this procedure.

INTRODUCTION

This paper will elaborate some of the lesser used functionalities of the Proc SQL procedure while providing industry context. The intention of this paper is to challenge the stigma attached to the usage of SQLs and encourage more commonplace implementation. To understand this paper, you will need a rudimentary understanding of SQL code and its basic applications.

CONDITIONAL DERIVATIONS

Proc SQLs allow you to apply conditional derivations, similar to the standard data-step “if… then…” statements. The syntax for which is “case… when… then…”. This can compress multiple steps into one.

Consider the following dataset for the examples in this section:

Figure 1: Dataset DEMOG, source dataset.

subject	sex	age
E1	F	20
E2	F	25
E3	F	30
E4	M	22
E5	M	23
E6	M	28

Example 1 – Simple conditional derivation:

```
proc sql;
  create table demog_agegrp as
  select *,
  case
    when age>=23 then 'Age >= 23'
    when age<23 then 'Age < 23'
  end as agegrp 'Age category' length=20
  from demog;
quit;
```

Note: The “” wildcard symbol selects all variables from the input dataset.*

Figure 2: Dataset DEMOG_AGEGRP, with derived variable AGRGRP.

subject	sex	age	agegrp
E1	F	20	Age < 23
E2	F	25	Age >= 23
E3	F	30	Age >= 23
E4	M	22	Age < 23
E5	M	23	Age >= 23
E6	M	28	Age >= 23

From the above code we can see the output dataset has been created while deriving the new variable AGEGRP and defining the length equal to 20; the variable label has also been set. The same could be accomplished using a SAS® data-step.

Example 2 - Conditional derivation with sort:

```
proc sql;
```

```
    create table demog_agegrp_ord as
    select *,
    case
       when age>=23 then 'Age >= 23'
       when age<23 then 'Age < 23'
    end as agegrp 'Age category' length=20
    from demog
    order by agegrp, sex, subject;
quit;
```

Figure 3: Dataset DEMOG_AGEGRP_ORD, with derived variable AGRGRP and sorting.

subject	sex	age	agegrp
E1	F	20	Age < 23
E4	M	22	Age < 23
E2	F	25	Age >= 23
E3	F	30	Age >= 23
E5	M	23	Age >= 23
E6	M	28	Age >= 23

Expanding on Example 1 this SQL is additionally sorting the output dataset by AGEGRP, SEX, SUBJECT. Now we start to see the value of SQL; this would require a data-step and a proc sort to accomplish with conventional code.

Example 3 - Conditional derivation with sort and merge:
For this example, the following dataset will also require consideration, this will be named *DEMOG_WGHT*:

Figure 4: Dataset DEMOG_WGHT.

subject	blwght
E1	50kg
E2	43kg
E4	70kg
E6	92kg
E7	66kg

```
proc sql;
    create table demog_agegrp_wght as
    select a.*, b.blwght,
    case
       when a.age>=23 then 'Age >= 23'
       when a.age<23 then 'Age < 23'
    end as agegrp 'Age category' length=20
    from demog as a
    left join demog_wght as b
    on a.subject=b.subject
    order by agegrp, sex, subject;
quit;
```

Figure 5: Dataset DEMOG_AGRGRP_WGHT, with derived variable AGRGRP and merge.

subject	sex	age	blwght	agegrp
E1	F	20	50kg	Age < 23
E4	M	22	70kg	Age < 23
E2	F	25	43kg	Age >= 23
E3	F	30		Age >= 23
E5	M	23		Age >= 23
E6	M	28	92kg	Age >= 23

Expanding even further on the previous examples, this SQL merges DEMOG with DEMOG_WGHT, derives AGEGRP and sorts the dataset by AGEGRP, SEX, SUBJECT. This would require multiple steps using conventional code but can be accomplished in one SQL. Note that subject E7 is not in *DEMOG_AGRGRP_WGHT since they are not found in DEMOG.*

METADATA REVIEW

SQLs can be used to explore the metadata of the libraries (and their contents) in an active SAS session. Though there are numerous SQL metadata references, the ones that will be covered are DICTIONARY.COLUMNS, TABLES. Their uses will be explored as well as comparison with the SASHELP.VCOLUMN, VTABLE that can be invoked within a SAS data-step. Updating metadata with the SQL “alter table” function will also be discussed. Note that the DICTIONARY datasets themselves cannot be altered, only the source domains.

SQL "DICTIONARY" LIBRARIES
Without specifying any conditions, the following calls will give the datasets as seen below from the DICTIONARY.COLUMNS, TABLES datasets respectively:

Note: The screenshots only show variables relevant for the forthcoming examples, they do not show the entire datasets.

Example 4 – DICTIONARY.COLUMNS:

```
proc sql;
  create table cols as
  select *
  from dictionary.columns;
quit;
```

Figure 6: Dataset COLS.

LIBNAME	MEMNAME	MEMTYPE	NAME	TYPE	LENGTH	NPOS	VARNUM	LABEL	FORMAT
WORK	DEMOG_AGEGRP_ORD	DATA	SUBJECT	char	200	8	1		
WORK	DEMOG_AGEGRP_ORD	DATA	SEX	char	200	208	2		
WORK	DEMOG_AGEGRP_ORD	DATA	AGE	num	8	0	3		
ADAM	ADSL	DATA	STUDYID	char	11	248	1	Study Identifier	
ADAM	ADSL	DATA	DOMAIN	char	2	259	2	Domain Abbreviation	
ADAM	ADSL	DATA	USUBJID	char	20	261	3	Unique Subject Identifier	
ADAM	ADSL	DATA	RANDDT	num	8	8	19	Date of Randomization	YYMMDD10.

Example 5 – DICTIONARY.TABLES

```
proc sql;
  create table dsets as
  select *
  from dictionary.tables;
quit;
```

Figure 7: Dataset DSETS.

LIBNAME	MEMNAME	MEMTYPE	MEMLABEL	CRDATE	NOBS	NVAR	MAXVAR	MAXLABEL
ADAM	ADAE	DATA	Adverse Events Analysis Dataset	29OCT19:15:29:39	56	141	8	40
ADAM	ADLB	DATA	Laboratory Test Results, Analysis Data	25JUL19:15:35:29	2042	138	8	40
ADAM	ADSL	DATA	Subject-Level Analysis Dataset	29OCT19:15:29:26	23	80	8	39
WORK	DEMOG_AGEGRP_ORD	DATA		03NOV19:18:17:02	6	4	7	12

You can see that the COLS dataset offers variable-level metadata, whereas the DSETS dataset concerns itself with domain-level metadata. Both can be very valuable during study work.

Consider the following scenarios:

Scenario 1
The variable APERIOD has been flagged upon review as not CDISC compliant. The study ADaM specifications are in poor condition and we need to know which ADaM domains will require an update.
The following code can determine which are the affected ADaM datasets:

```
proc sql;
  create table per_doms as
  select memname
  from dictionary.columns
  where libname='ADAM' and name='APERIOD';
quit;
```

This code will read in the COLUMNS dataset and return the dataset names (MEMNAME) which contain a variable name (NAME) of APERIOD. The output dataset will indicate the domains requiring updates to fix the issue with APERIOD.

Note: You will observe in COLS, DSETS that MEMTYPE="DATA" for all records. For simplicity, this subsetting will not be required for these examples. However, metadata is collected for items other than datasets. Therefore, in general it is good practice to add "`memtype='DATA'`*" to the where clauses to improve processing efficiency.*

Scenario 2

It has been identified as a quality issue that variable labels have regularly been inconsistent across ADaM domains for a particular sponsor. It has been requested that all ongoing studies for the concerned sponsor program checks for this issue.

The following code will count the number of unique label values in the ADaM library across all datasets, returning records that have more than one label per variable:

```
proc sql;
  create table uniq_ad_label as
  select distinct name, label
  from dictionary.columns
  where libname='ADAM';

  create table mult_label as
  select name, label
  from uniq_ad_label
  group by name
  having count(*)>1;
quit;
```

Note: The "having" clause will be discussed further in the subsequent section.

Figure 8: Dataset MULT_LABEL, indicating variables with inconsistent variable labels.

	NAME	LABEL
1	ADTM	Analysis Date/Time
2	ADTM	Analysis Datetime

We can see from the output dataset that the only variable with a label inconsistency is ADTM. We can then adapt the code from scenario 1 to determine the concerned domains using ADTM.

Scenario 3

A new data-cut for your study was received on 22Aug2019. You want to ensure that that all ADaM programs have been rerun since the new data was received.

The following code will use the TABLES dictionary (Figure 7) to check the creation date of all datasets within the ADaM library and will return any which were created prior to the new data-cut:

```
proc sql;
  create table dset_date as
  select memname
  from dictionary.tables
  where libname='ADAM' and datepart(crdate)<'22AUG2019'd;
quit;
```

We can see from Figure 7 that ADLB has not been run since the new data was received. Therefore, the above SQL code will pull out this dataset value.

Note: The above code can be adapted to check if the timestamp of the QC datasets are after production as anticipated.

Scenario 4

We want to create a macro variable containing the number of observations for a certain dataset. Subsequent macro programming will account for the cases where the number of observations are 0 and >0 respectively.

Note: For demonstration purposes the below example is concerned with the data shown in Figure 1.

```
proc sql;
  select nobs
  into: numobs trimmed
  from dictionary.tables
  where libname='WORK' and memname='DEMOG';
quit;
```

The macro variable "numobs" is created with the value of 6.

DATA-STEP "SASHELP" LIBRARY COMPARISON

Everything demonstrated in the previous section can also be accomplished using SAS data-steps; replacing DICTIONARY with SASHELP and COLUMNS, TABLES with VCOLUMN, VTABLE respectively. The following code can create the same dataset as PER_DOMS (Scenario 1) using a data-step:

```
data per_doms_alt(keep=memname);
  set sashelp.vcolumn(keep=memname libname name);
  where libname='ADAM' and name='APERIOD';
run;
```

The main difference of concern between these two methods is the processing time. Consider the SQL call given in Scenario 1. The processing time for this can be observed:

```
NOTE: PROCEDURE SQL used (Total process time):
      real time           0.09 seconds
      cpu time            0.00 seconds
```

Comparatively, PER_DOMS_ALT results in the following processing time:

```
NOTE: DATA statement used (Total process time):
      real time           4.60 seconds
      cpu time            1.40 seconds
```

This is a substantial difference is processing time between the two methods to accomplish the same result. Hence, using SQL code has a clear benefit in this instance. Note, part of the speed gain is due to the code optimiser active within SQL.

UPDATING METADATA

The “alter table” function offers an alternative way to redefine variable metadata. Consider the following metadata sample from the ADSL domain:

Figure 9: DICTIONARY.COLUMNS ADSL metadata snippet.

NAME	TYPE	LENGTH	LABEL	FORMAT
RACE	char	100	Race	
STUDYID	char	11	Study Identifier	
USUBJID	char	20	XXXXXX	
RANDFL	char	2	Randomized Population Flag	
RANDDT	num	8	Date of Randomization	YYMMDD10.

We can see that there is an issue with the label for USUBJID. Additionally, we have also identified that the length for RACE is too long and needs to be decrease to 10. Lastly, it has been requested that RANDDT be presented in DATE9. format. The following SQL code can be used:

```
proc sql;
  alter table adsl
  modify race char(10), usubjid label='Unique Subject Identifier', randdt format=date9.;
quit;
```

Figure 10: DICTIONARY.COLUMNS ADSL metadata snippet, following ADSL metadata update.

NAME	TYPE	LENGTH	LABEL	FORMAT
STUDYID	char	11	Study Identifier	
USUBJID	char	20	Unique Subject Identifier	
RACE	char	10	Race	
RANDFL	char	2	Randomized Population Flag	
RANDDT	num	8	Date of Randomization	DATE9.

We can see from the metadata that the dataset has been updated as intended. Of course, the same can be accomplished within a data-step using an “attrib” statement or using Proc Datasets (though this only allows modification of labels and formats, not lengths). The difference in processing time is nominal, this is merely intended to showcase an additional SQL functionality and alternative approach to updating metadata.

RUDIMENTARY CALCULATIONS

SQL code can also be used as an alternative to a proc sort (both a standard and a distinct sort). Further, SQLs can be practical for simple calculation (such as min, max and frequency) when applied in conjunction with the “group by” clause. The “having” clause is also useful when used with the “group by” clause. This will be elaborated, as well as the distinction between “having” and “where” clauses being made.

SQL SORTS

Consider the following code:

```
proc sql;
  create table demog_filt_srt as
  select age, subject
  from demog
```

```
    where age>22
    order by age;
  quit;
```

This reads in the DEMOG dataset, applies a condition where AGE is greater than 22, sorts by AGE and outputs the DEMOG_FILT_SRT dataset. This code can be broken down analogously with standard data-step, proc sort terminology:

SQL	Data-step and Proc Sort
Create table demog_filt_srt as	Data demog_filt_srt;
Select age, subject	Set demog(keep=age subject where=(age>22));
From demog	Set demog(keep=age subject where=(age>22));
Where age>22	Set demog(keep=age subject where=(age>22));
Order by age	Proc sort data=demog_fil_srt; by age; run;

You can see the “order by” clause can be considered as equivalent to the standard “proc sort”. Additionally, if we have a dataset containing duplicates the “distinct” option can be added to the “select” clause to create a unique list:

```
  proc sql;
    create table demog_filt_srt_uniq as
    select distinct age, subject
    from demog
    where age>22
    order by age;
  quit;
```

The addition of “distinct” to the select clause tells the SQL to only return unique or “distinct” observations for the combination of AGE and SUBJECT. The “order by” clause then functions as before, sorting by the variables indicated in this clause. Specifying “distinct” can be seen as the equivalent as including the “nodupkey” option within a proc sort.

A few points to note:

1. When specifying “distinct”, the “order by” clause is not strictly necessary. In this case the SQL will sort the dataset in the order the variables in the “select distinct” clause were specified. The “order by” clause could still be useful in cases where you want to sort the output dataset differently to that given in the “select distinct” clause.
2. The output dataset will only keep variables specified in the “select distinct” clause and likewise you can only use these variables in the “order by” clause. In contrast a proc sort nodupkey will keep all variables, unless otherwise specified in a “keep”/”drop” statement. This is a limitation of the SQL code.

“GROUP BY” CLAUSE

For simple dataset creation the “group by” clause acts much the same as the “order by” clause. You see that if the “order by” code is replaced as indicated we will get the following log note:

Order by	Group by
proc sql; create table demog_filt_srt as select age, subject from demog where age>22 order by age; quit;	proc sql; create table demog_filt_srt as select age, subject from demog where age>22 group by age; quit;

```
WARNING: A GROUP BY clause has been transformed into an ORDER BY clause because neither
the SELECT clause nor the optional HAVING clause of the associated table-expression referenced a summary function.
```

This log note alludes that some form of “summary function” must be used in the SQL for the “group by” clause to be relevant. If you are unsure what a “summary function” is, the majority of univariate summary statistic qualify. For the following, let’s return to the example data given in Figure 5. The following SQL will generate various summary statistic, stratifying (or grouping) by sex:

```
  proc sql;
    create table demog_sum_stat as
    select *,
      count(age) as n,
      mean(age) as mean,
      median(age) as med,
```

```
    min(age) as min,
    max(age) as max
  from demog_agegrp_wght
  group by sex;
quit;
```

Figure 11: Dataset DEMOG_SUM_STAT, with summary statistics.

SUBJECT	SEX	AGE	BLWGHT	AGEGRP	N	MEAN	MED	MIN	MAX
E1	F	20	50kg	Age < 23	3	25	25	20	30
E2	F	25	43kg	Age >= 23	3	25	25	20	30
E3	F	30		Age >= 23	3	25	25	20	30
E6	M	28	92kg	Age >= 23	3	24.333333333	23	22	28
E4	M	22	70kg	Age < 23	3	24.333333333	23	22	28
E5	M	23		Age >= 23	3	24.333333333	23	22	28

Now we have instructed the SQL to create summary statistics the “group by” clause runs without warning and we can see these new variables have been stratified by SEX as specified.

We can see the statistics are populated against all records in the input dataset since “*” was indicated in the “select” clause. If we wanted to do this with conventional methods, a proc mean/univariate would be required followed by a merging data-step. If we only want one record per SEX we simply need replace “*” with “SEX” in the “select” clause.

Additionally, if we want the data ordered differently from how the summary statistics are stratified, we can add an “order by” clause after “group by”.

“HAVING” CLAUSE

As the log warning in the previous section suggested, the “having” clause can also be used in the application of summary statistic generation. This function works like a “where” clause. However, there a few fundamental differences:

1. Unless additional code is added, the “where” clause will apply the conditions to the input dataset. “having” will apply the conditions to the output dataset. Therefore, the “where” clause will not work if you refer to variables created within the concerned SQL. The “having” clause will.
2. “having” clauses allow the use of summary functions within the clause. This is not possible in “where” clauses as they process the conditions specified at a record level rather than considered the data collectively as specified in the “group by” clause. This should become clearer in the following examples.

Now suppose we are only concerned with records where the age is equal to the minimum age found across all subjects:

```
proc sql;
  create table demog_mage as
  select *
  from demog_agegrp_wght
  having age=min(age);
quit;
```

Figure 12: Dataset DEMOG_MAGE, keeping minimum age.

SUBJECT	SEX	AGE	BLWGHT	AGEGRP
E1	F	20	50kg	Age < 23

We can see that the record with the overall minimum value has been returned. If we want the minimum value per SEX, we only need include the “group by” clause and specify the stratifying variable:

```
proc sql;
  create table demog_mage_grp as
  select *
  from demog_agegrp_wght
  group by sex
  having age=min(age);
quit;
```

Figure 13: Dataset DEMOG_MAGE_GRP, keeping minimum age by SEX.

SUBJECT	SEX	AGE	BLWGHT	AGEGRP
E1	F	20	50kg	Age < 23
E4	M	22	70kg	Age < 23

Comparatively, if we were to replace “having” with “where” in the above code we would get the following log note:

```
ERROR: Summary functions are restricted to the SELECT and HAVING clauses only.
```

In work related applications, “having” clauses can be used to determine things such as baseline or critical values. The code below was used to determine the maximum post-baseline values per laboratory parameter in the ADLB domain. Comparison of this code against conventional methods is also indicated:

Note: In the study this example was taken from, baseline reading was taken upon first dose, there were no pre-dose readings.

SQL method:

```
proc sql;
  create table max_lab as
  select distinct usubjid, param, aval
  from adlb
  where ablfl^="Y" and aval^=.
  group by usubjid, param
  having aval=max(aval);
quit;
```

Conventional method:

```
proc sort data=adlb(keep=usubjid param aval ablfl where=(ablfl^="Y" and aval^=.))
          out=mlab(drop=ablfl);
  by usubjid param aval;
run;

data mlab2;
  set mlab;
  by usubjid param aval;
  if last.param;
run;
```

Though, the conventional method is not complicated, it is clearly not as elegant and requires two steps rather than the one required using the SQL method. Combining many of the SQL functionalities discussed in the paper, this SQL example:

1. Filters the input dataset for post baseline non-missing results.
2. Creates a unique list of records by subject, laboratory parameter, laboratory result. Keeping only USUBJID, PARAM, AVAL in the output dataset.
3. Outputs 1 result per subject per laboratory parameter, where AVAL is the maximum result within these “by” groups.

Though, the conventional method by no means is inefficient and is still very useful, this should demonstrate the utility and versatility that SQL code can offer.

CONCLUSION

This paper has demonstrated the versatility and strengths of SQLs while only focusing on a snapshot of SQL functionalities. Additionally, comparisons against conventional programming have been offered and the respective advantages of SQL code showcased. Hopefully, this paper has challenged the negative stigma associated with using SQL code in our industry and encouraged programmers to use this more regularly in the future.

CONTACT INFORMATION

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

Author Name: Joseph Murphy
Company: IQVIA
Address: 3 Forbury Place, 23 Forbury Road
City / Postcode: Reading, United Kingdom, RG1 3JH
Work Phone: +441184501148
Email: joseph.murphy@iqvia.com