Paper 3154-2015

PROC SQL for PROC SUMMARY Stalwarts

Christianna Williams PhD, Chapel Hill, NC

ABSTRACT

One of the fascinating features of SAS® is that the software often provides multiple ways to accomplish the same task. A perfect example of this is the aggregation and summarization of data across multiple rows or BY groups of interest. These groupings can be study participants, time periods, geographical areas, or really just about any type of discrete classification that one desires. While many SAS programmers may be accustomed to accomplishing these aggregation tasks with PROC SUMMARY (or equivalently, PROC MEANS), PROC SQL can also do a bang-up job of aggregation – often with less code and fewer steps. This step-by-step paper explains how to use PROC SQL for a variety of summarization and aggregation tasks. It uses a series of concrete, task-oriented examples to do so. For each example, both the PROC SUMMARY method and the PROC SQL method will be presented, along with discussion of pros and cons of each approach. Thus, the reader familiar with either technique can learn a new strategy that may have benefits in certain circumstances. The presentation format will be similar to that used in the author's previous paper, "PROC SQL for DATA Step Die-Hards".

INTRODUCTION

Descriptive analytic and data manipulation tasks often call for aggregating or summarizing data in some way, whether it is calculating means, finding minimum or maximum values or simply counting rows within groupings of interest. Anyone who has been working with SAS for more than about two weeks has probably learned that there are nearly always about a half dozen different methods for accomplishing any type of data manipulation in SAS (and often a score of user group papers on these methods ☺) and summarizing data is no exception. The two primary and most flexible strategies for aggregating data are probably PROC SUMMARY and PROC SQL, and many SAS programmers are comfortable with one technique but not the other. The purpose of this paper is to serve as a sort of "cross-walk" between PROC SUMMARY and PROC SQL, demonstrating how each can be used to accomplish a series of aggregation tasks. The examples are explained in substantial detail, with variations and coding tips sprinkled throughout. By demonstrating what is "easy" to do with one method, and more convoluted with the other, I hope to give you the toolkit to make an informed choice of methods for different types of tasks – even if that takes you out of your current programming "comfort" zone. The details can be important, and differences in the behavior of the two methods in some areas (for example, the handling of missing data on grouping variables) could cause unexpected results for the uninformed or unwary programmer.

I wanted to make the examples quasi-realistic (and to use data with which I was already familiar); thus, all of the examples are based on a data set containing demographic and health information on the approximately 1.4 million residents of US nursing homes in the fourth quarter of 2012. These data (in many different summary forms, thanks to PROC SQL and PROC SUMMARY!) are reported in the 2013 edition of the Centers for Medicare and Medicaid Nursing Home Data Compendium (CMS, 2014).

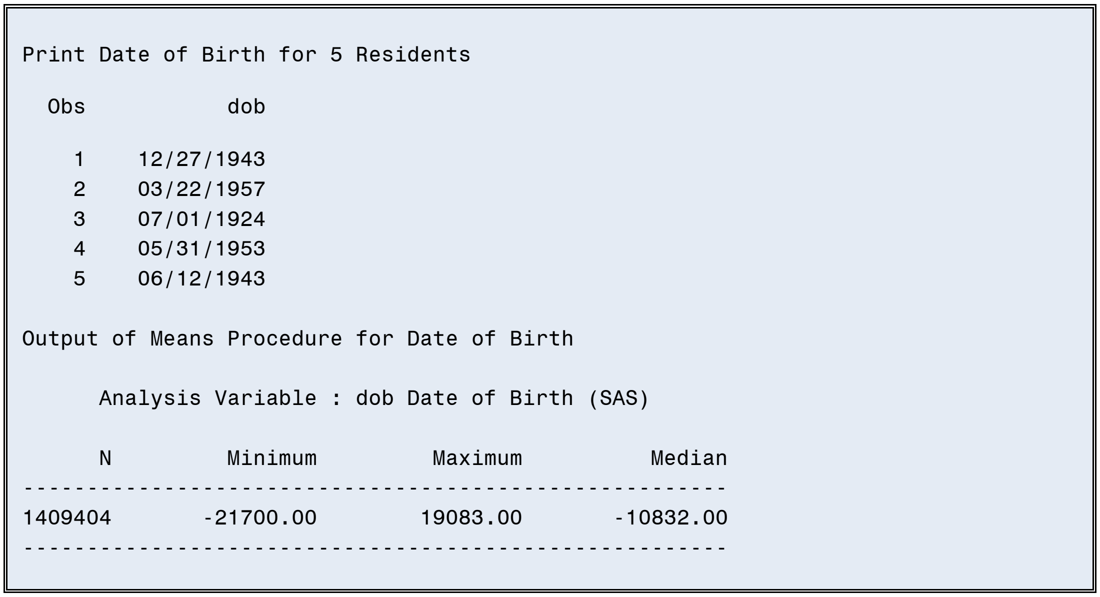
EXAMPLE 1: SUMMARIZE A SINGLE MEASURE OVER ALL ROWS

Let's say we want to get the earliest, latest and median date of birth for all the nursing home residents in the country. While a birth date is a bit of an odd type of measure to summarize, I use it because it helps to demonstrate some of the less obvious differences between PROC SUMMARY and PROC SQL. On my data file the variable DOB is the resident's birth date, and it is a SAS date formatted as MMDDYY10. (i.e. mm/dd/yyyy). SAS dates, of course, are integers, which are the number of days since January 1, 1960. Assigning them a FORMAT (from which there are many to choose!) has NO effect on the underlying value of the variable, but it does make them easier for most humans to read.

For most examples, in this paper, I'm going to use PROC SUMMARY because it is set up to produce an output data set, and that is what I usually want to do. However, here, I'm going to first use PROC MEANS to show one potentially important difference between MEANS and SUMMARY.

```
PROC MEANS DATA = Residents2012 N MIN MAX MEDIAN ;
VAR dob ;
RUN;
```

Output 1 lists the birth dates for 5 residents in the file (to prove the *dob* variable is formatted), and then shows the PROC MEANS result. You can see that even though the original variable is formatted, the statistics in the output or not. Adding a FORMAT statement to the PROC MEANS code above has no effect. Also, while we could use the MAXDEC= option or the FW= option to control the way the statistics are printed (e.g. removing decimal places), they still would be easily seen as dates.



```
Print Date of Birth for 5 Residents

  Obs           dob

    1    12/27/1943
    2    03/22/1957
    3    07/01/1924
    4    05/31/1953
    5    06/12/1943

Output of Means Procedure for Date of Birth

      Analysis Variable : dob Date of Birth (SAS)

      N         Minimum         Maximum          Median
-------------------------------------------------------
1409404       -21700.00        19083.00       -10832.00
-------------------------------------------------------
```

Output 1 Even though the date of birth variable has a date format on the raw data, this is lost in the PROC MEANS Output.

However, we get a pleasant surprise if we use PROC SUMMARY:

```
PROC SUMMARY DATA = Residents2012 ;
VAR dob ;
OUTPUT OUT = dobsum1
         N = numdob
       MIN = mindob
       MAX = maxdob
    MEDIAN = meddob   ;
RUN;

PROC PRINT DATA = dobsum1 NOOBS;
TITLE 'Output of SUMMARY Procedure for Date of Birth';
RUN;

PROC PRINT DATA = dobsum1 NOOBS LABEL;
TITLE 'Output of SUMMARY Procedure for Date of Birth (with LABEL option)';
RUN;
```

The PROC PRINT output is shown in ***Output 2***, and we see that PROC SUMMARY assigns the FORMATs of the underlying variable to the summary statistics, which is helpful here. It is a little less fortunate that other variable attributes (namely LABELs) are also assigned to all the summary statistic variables, as shown in the bottom part of ***Output 2***. This is even the case for the *N* variable. There is a way around this, which I'll come back to in a minute. One final note on the inheritance of attributes of the raw variables in PROC SUMMARY: variable numeric LENGTH is NOT inherited – the summary stats will have a numeric length of 8 bytes, regardless of the numeric length of the variable summarized. This is a good thing in order to maintain precision – it might be OK to store small integers with a length of 3 or 4, for example, but we probably wouldn't be happy if the mean or standard deviation of such a variable was shoehorned into that little space!!

Another feature of the PROC SUMMARY output is the automatic variables *_TYPE_* and *_FREQ_.* The *_TYPE_* indicates the level of aggregation, and can be very useful when multiple CLASS variables are specified, but is beyond the scope of this paper. The *_FREQ_* variable tells us how many observations were aggregated in producing the summary statistics – here because we are aggregating over the entire file (i.e. no CLASS variables), it tells us the number of observations in the entire data set. The difference between this and the N variable (*numdob*), tells us how many observations have missing data for date of birth.

```
Output of SUMMARY Procedure for Date of Birth

_TYPE_     _FREQ_     numdob        mindob        maxdob        meddob

   0      1409749    1409404    08/03/1900    03/31/2012    05/06/1930

Output of SUMMARY Procedure for Date of Birth (with LABEL option)

                     Date of       Date of       Date of       Date of
                      Birth         Birth         Birth         Birth
_TYPE_     _FREQ_     (SAS)         (SAS)         (SAS)         (SAS)

   0      1409749    1409404    08/03/1900    03/31/2012    05/06/1930
```

Output 2. Using PROC SUMMARY and then PROC PRINT we see that the aggregates retain the FORMATs (and LABELs) of the raw data.

Presumably you are reading this paper because you want to learn about summarizing data with PROC SQL, and here we are already on page 3 with 'nary a SELECT clause to be seen. So, before I tell you how to get more informative labels on the PROC SUMMARY output variables, let's demonstrate how to use PROC SQL to summarize a single measure across the entire input file. In this code, we are not creating a new data set – the result will go to whatever output destinations are active. We are using summary functions that have very familiar names.

```
PROC SQL ;
SELECT N(dob) AS dob_N
      ,MIN(dob) AS dob_Min
      ,MAX(dob) AS dob_Max
      ,MEDIAN(dob) AS dob_Median
FROM Residents2012;
QUIT;
```

Output 3 shows the result.

```
dob_N   dob_Min   dob_Max  dob_Median
---------------------------------------
 1409404    -21700     19083      -10832
```

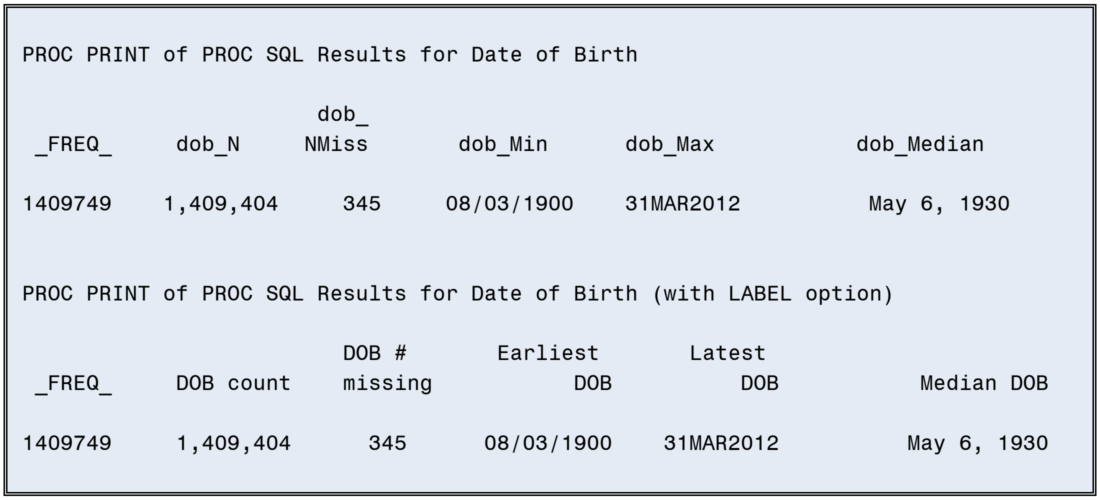
Output 3 .Using PROC SQL to aggregate date of birth, the FORMAT of the underlying variable is not inherited.

Immediately we see that the date format is not retained in the output. The same would be true even if we inserted a CREATE TABLE clause before the SELECT above. And we must give the columns names (e.g. the “AS *dob_N*” syntax above). Otherwise, the output would have NO headers on the columns (try it at home!), and if you were creating a data set and did not assign column names (aka “aliases”), SAS would make up names for the new variables, and I guarantee you that you would not be happy with the result. The names – at least on my machine – come out as things like *_TEMG001*, and there is nothing in the log to tell you that you have done something stupid!

Also note that the *N* variable is counting the number of non-missing values for its argument – it gives the same result as the N statistic in PROC SUMMARY/MEANS. We can use the COUNT(*) syntax to get the number of rows aggregated and the NMISS function to explicitly get the number of rows with missing data for the analysis variable. This is shown below, along with the way to add FORMATs to the summary statistics. Just to show that you can, I use different FORMATs for the different functions. This is not possible with PROC SUMMARY without additional processing (i.e. a subsequent DATA step). Note that I am also giving distinct LABELs to each of the new aggregate columns.

```
PROC SQL ;
CREATE TABLE dobsum2 AS
SELECT COUNT(*) AS _FREQ_
      ,N(dob)    AS dob_N      LABEL ='DOB count'  FORMAT=COMMA10.
      ,NMISS(dob) AS dob_NMiss LABEL='DOB # missing'
      ,MIN(dob)  AS dob_Min    LABEL = 'Earliest DOB' FORMAT=MMDDYY10.
      ,MAX(dob)  AS dob_Max    LABEL = 'Latest DOB'   FORMAT=DATE9.
     ,MEDIAN(dob) AS dob_Median LABEL ='Median DOB' FORMAT=WORDDATE32.
FROM Residents2012;
QUIT;
```

Output 4 shows the result, both with and without the LABELs.



```
PROC PRINT of PROC SQL Results for Date of Birth

                         dob_
 _FREQ_     dob_N      NMiss       dob_Min      dob_Max          dob_Median

1409749    1,409,404     345     08/03/1900    31MAR2012          May 6, 1930


PROC PRINT of PROC SQL Results for Date of Birth (with LABEL option)

                           DOB #       Earliest       Latest
 _FREQ_     DOB count    missing            DOB          DOB          Median DOB

1409749     1,409,404      345      08/03/1900    31MAR2012         May 6, 1930
```

Output 4. With PROC SQL, you can easily assign desired FORMATs and LABELs to the summary statistics

We saw earlier that by default PROC SUMMARY assigns the same FORMAT and LABEL to all the summary statistics, which may not be ideal. However, there are a couple of very handy options to at least solve the LABEL issue. The AUTONAME option saves you the trouble of giving names to each of the output statistics and AUTOLABEL provides distinct and informative LABELs. You can use these options alone or together.

```
PROC SUMMARY DATA = Residents2012 ;
VAR dob ;
OUTPUT OUT = dobsum1A
         N =
      NMISS =
       MIN =
      MAX =
      MEDIAN = / AUTONAME AUTOLABEL;
RUN;
```

Output 5 shows the result, both with and without the LABELs.

```
Output of SUMMARY Procedure for Date of Birth with AUTONAME and AUTOLABEL options

                                 dob_
_TYPE_     _FREQ_     dob_N     NMiss       dob_Min       dob_Max    dob_Median

   0      1409749    1409404     345      08/03/1900    03/31/2012    05/06/1930


Output of SUMMARY Procedure for Date of Birth with AUTONAME and AUTOLABEL options
Printing with LABEL option

                                Date of                                   Date of
                     Date of     Birth_       Date of       Date of        Birth_
_TYPE_     _FREQ_    Birth_N     NMiss      Birth_Min     Birth_Max        Median

   0      1409749    1409404      345      08/03/1900    03/31/2012    05/06/1930
```

Output 5. The AUTONAME option of PROC SUMMARY gives distinct, informative names to the summary statistics requested, and AUTOLABEL provides meaningful labels.

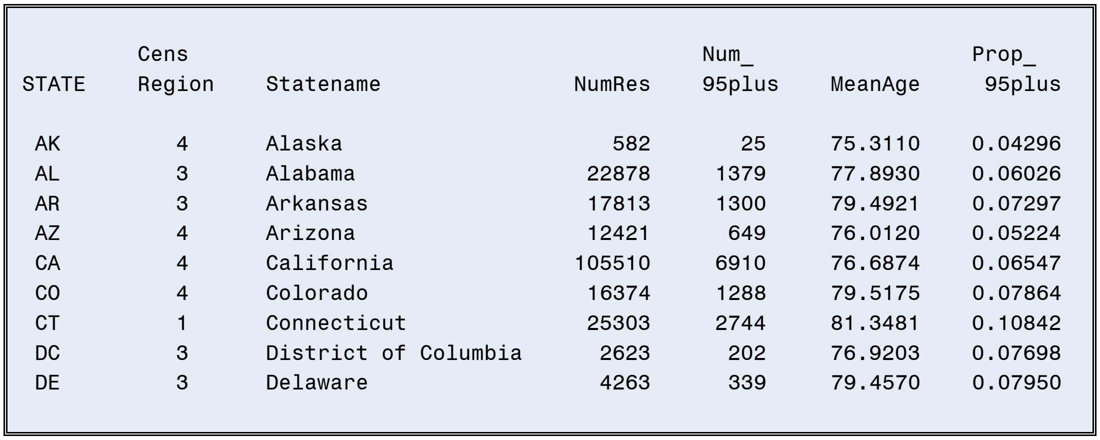
So, which is better for this purpose – SUMMARY or SQL? Well, as usual, I think it depends on the specifics of the task. The AUTONAME and AUTOLABEL options in SUMMARY are very handy – if the naming/labeling conventions are acceptable. Note that If the LABEL of the original variable is long, then it may get truncated by the AUTOLABEL option. It is also a nice feature of SUMMARY that the FORMAT of the original variable is inherited – however, if you ever need to give different FORMATs to different statistics, PROC SQL is a little simpler. In summary, PROC SQL gives you more flexibility, but for this simple task, may require a little more code.

EXAMPLE 2: SUMMARIZE MORE THAN ONE MEASURE FOR SPECIFIED GROUPS

Ok…let's move on to a slightly more complex example. Let's say that we to aggregate several measures by state. Here we use two variables – *Age*, which is an integer value corresponding to the resident's age in years and *Age_ge95*, which is a dichotomous indicator that has value 1 if the resident is age 95 or older and 0 otherwise. The following PROC SUMMARY step counts the number of residents per state with non-missing age [N(*Age*) = *NumRes*]; counts the number of residents per state who are 95 years old or older [SUM(*Age_ge95*) = *Num_95plus*] calculates the average resident age and the proportion who are 95 of greater [MEAN = *MeanAge Prop_95plus*]. Of course, the CLASS statement means that these summarizations will be done separately for each state (the *STATE* variable is the two-letter postal abbreviation for each state); hence the data set *AGESUM1* will have one observation per state (plus District of Columbia), while the NWAY option on the PROC SUMMARY statement means that there will be no overall row in the output. Unlike with a BY statement, the data set does not need to be sorted by the variable(s) listed on the CLASS statement. The DROP = _: data set option on the *AGESUM1* data set will remove the variables in the output that start with underscore – in this case, it is a shorthand way of eliminating the automatic *_TYPE_* and *_FREQ_* variables. Finally, the ID statement will add the census region (1-4) and the name of the state to each row in the output. The ID statement is useful to add columns that are at the same level of aggregation as the CLASS variable (as in the case of *StateName*) or higher (as in the case of *CensRegion*); otherwise the ID variable values will simply be the values of these variables on the last observation in the input data for each level of the CLASSification variables, which could be very misleading if those values had not been constant across all observations within a category of the CLASS variables.

```
PROC SUMMARY DATA = Residents2012 NWAY;
CLASS state ;
VAR age Age_ge95;
OUTPUT OUT = agesum1 (DROP = _:)
         N(age) = NumRes
     SUM(Age_ge95) = Num_95plus
        MEAN = MeanAge Prop_95plus ;
ID CensRegion StateName ;
RUN;
```

The first 10 rows of the *AGESUM1* data set (which has 51 total rows) are shown in ***Output 6***.



```
        Cens                                       Num_                 Prop_
STATE   Region    Statename               NumRes   95plus    MeanAge    95plus

 AK       4       Alaska                     582      25     75.3110   0.04296
 AL       3       Alabama                  22878    1379     77.8930   0.06026
 AR       3       Arkansas                 17813    1300     79.4921   0.07297
 AZ       4       Arizona                  12421     649     76.0120   0.05224
 CA       4       California              105510    6910     76.6874   0.06547
 CO       4       Colorado                 16374    1288     79.5175   0.07864
 CT       1       Connecticut              25303    2744     81.3481   0.10842
 DC       3       District of Columbia      2623     202     76.9203   0.07698
 DE       3       Delaware                  4263     339     79.4570   0.07950
```

Output 6. Partial listing of PROC SUMMARY output data set after aggregation of two measures by state.

Note that the output data is sorted by the CLASS variable – this will happen regardless of whether the input data was sorted. Also note that the order of the columns on the output data set is first the class variables, then the ID variables (if any) and then the requested statistics – in the order they are specified on the OUTPUT statement.

The PROC SQL code to achieve a similar result is as follows:

```
PROC SQL ;
CREATE TABLE agesum2 AS
SELECT state
      ,N(age) AS NumRes
     ,SUM(Age_ge95) AS Num_95plus
     ,MEAN(Age) AS MeanAge
     ,MEAN(Age_ge95) AS Prop_95plus
FROM Residents2012
GROUP BY state
ORDER BY state ;
QUIT;
```

The GROUP BY clause functions similarly to the PROC SUMMARY CLASS variable – it dictates the groupings on which the SUMMARY functions operate. The ORDER BY clause is not required, but it specifies the sort order of the output table *AGESUM2*. There is nothing to prevent you from ordering the new table by some other column (which would not be possible with PROC SUMMARY – it would require a separate PROC SORT step. Note that the code does not SELECT *Statename* or *CensRegion* so these columns will not be on the new table. I will come back to this point in a moment, but note that the only columns that are SELECTED are either the GROUP BY variable or a summary function.

The first 10 rows of the *AGESUM2* data set are shown in ***Output 7***. This is identical to the PROC SUMMARY output above except for the absence of *CensRegion* and *Statename*. Note that I could easily have added FORMATS and LABELS to the new columns, as shown in Example 1 if desired.

```
                    Num_                  Prop_
STATE    NumRes    95plus    MeanAge     95plus

 AK         582        25    75.3110    0.04296
 AL       22878      1379    77.8930    0.06026
 AR       17813      1300    79.4921    0.07297
 AZ       12421       649    76.0120    0.05224
 CA      105510      6910    76.6874    0.06547
 CO       16374      1288    79.5175    0.07864
 CT       25303      2744    81.3481    0.10842
 DC        2623       202    76.9203    0.07698
 DE        4263       339    79.4570    0.07950
 FL       76261      5554    79.2542    0.07281
```

Output 7. Partial listing of PROC SQL output data set after aggregation of two measures by state.

The most obvious way to add *CensRegion* and *Statename* to the summary file generated by PROC SQL – just adding them to the SELECT clause as shown below – does not work as expected.

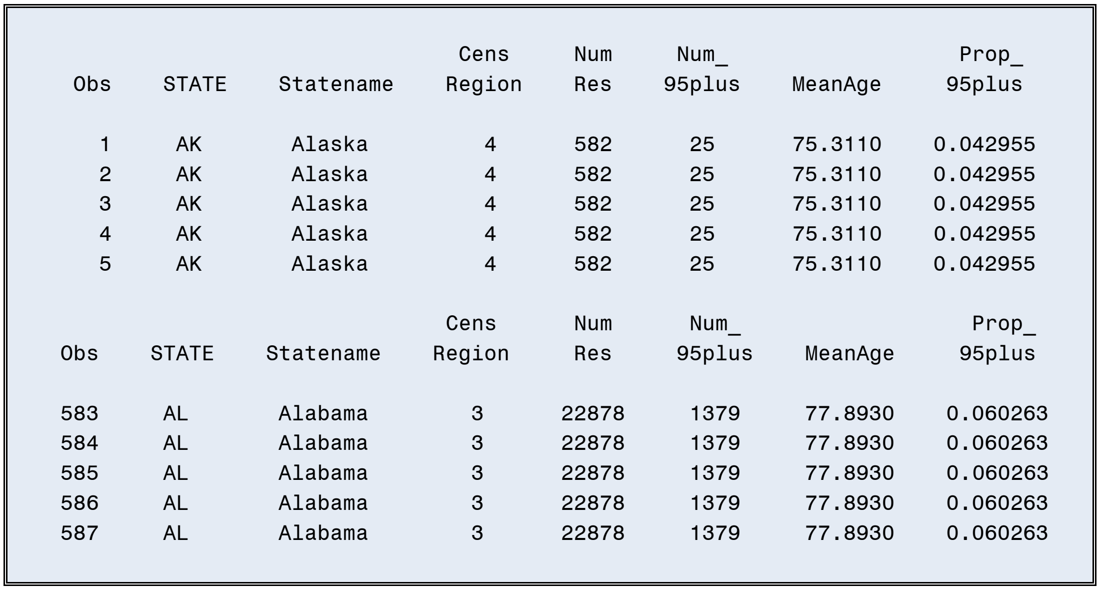
```
PROC SQL ;
CREATE TABLE NotWhatWeWant AS
SELECT state
      ,StateName
      ,CensRegion
      ,N(age) AS NumRes
     ,SUM(Age_ge95) AS Num_95plus
     ,MEAN(Age) AS MeanAge
     ,MEAN(Age_ge95) AS Prop_95plus
FROM Residents2012
GROUP BY state
ORDER BY state ;
QUIT;
```

We do not get an error, but we get the following notes in the SAS log (***Output 8***):

```
NOTE: The query requires remerging summary statistics back with the original data.
NOTE: Table WORK.NOTWHATWEWANT created, with 1409749 rows and 7 columns.
```

Output 8. Log Notes after adding Statename and CensRegion – which are neither GROUP BY variables nor functions of GROUP BY variables to the SELECT clause.

The issue is that *Statename* and *CensRegion* are neither on the GROUP BY clause, nor are they functions of the GROUP BY variables. And PROC SQL doesn't "know" that these variables happen to have identical values for all rows in a given level of the GROUP BY variables – that is that they are at the same or a higher level of aggregation. So, in order to carry out the query without losing information, SQL essentially merges (or 're-merges') the summary state-level data table with the resident-level table. ***Output 9*** lists the first 5 observations for each of Alaska and Alabama – the rows within each state are identical only because the values of *Statename* and *CensRegion* are, of course, identical within a state. If other columns were in the SELECT that were not identical across rows in the input *Residents2012* data set (e.g. *Age* itself), the re-merge would still happen and these values would differ from row to row as on the un-summarized file.



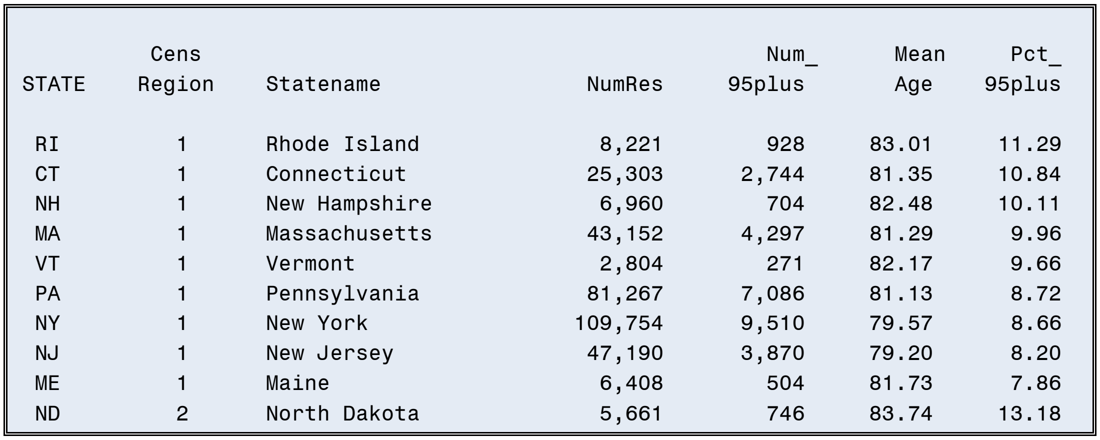
```
                                Cens     Num     Num_                 Prop_
  Obs    STATE    Statename    Region    Res    95plus    MeanAge     95plus

    1     AK       Alaska         4      582      25      75.3110    0.042955
    2     AK       Alaska         4      582      25      75.3110    0.042955
    3     AK       Alaska         4      582      25      75.3110    0.042955
    4     AK       Alaska         4      582      25      75.3110    0.042955
    5     AK       Alaska         4      582      25      75.3110    0.042955

                                Cens     Num     Num_                  Prop_
 Obs    STATE    Statename    Region     Res    95plus    MeanAge      95plus

 583     AL       Alabama        3      22878    1379     77.8930    0.060263
 584     AL       Alabama        3      22878    1379     77.8930    0.060263
 585     AL       Alabama        3      22878    1379     77.8930    0.060263
 586     AL       Alabama        3      22878    1379     77.8930    0.060263
 587     AL       Alabama        3      22878    1379     77.8930    0.060263
```

Output 9. Partial listing of PROC SQL output data set "re-merge"

There are a few different ways to achieve the desired result here – getting these two additional columns onto the summary data set without generating all these additional rows – but the most efficient way is to simply add them to the GROUP BY clause. Recall that the data set does not need to be sorted by the GROUP BY variables. This is shown in the code below. I've also demonstrated a few other possible refinements. I've added FORMATs to each of the summary variables. Note that I also multiply the MEAN of the Boolean *Age_ge95* variable by 100 to convert it from a proportion (0-1) to a percent (0-100). Finally, as I noted earlier, I am specifying different columns on the ORDER BY clause than those on from the GROUP BY. This does not affect the aggregation at all – it simply specifies the SORT order of the resulting data set. Unlike in the BY statement of the DATA Step, or PROC SORT or most (all?) other SAS procedures, the DESCENDING keyword (specifying that the ordering will be from largest to smallest) goes after the column name that it modifies.

```
PROC SQL ;
CREATE TABLE agesum2b AS
SELECT CensRegion
      ,state
      ,statename
      ,N(age) AS NumRes FORMAT=COMMA9.
     ,SUM(Age_ge95) AS Num_95plus FORMAT=COMMA7.
     ,MEAN(Age) AS MeanAge FORMAT=6.2
     ,100*MEAN(Age_ge95) AS Pct_95plus FORMAT=6.2
FROM Residents2012
GROUP BY CensRegion, state, statename
ORDER BY CensRegion, Pct_95plus DESCENDING   ;
QUIT;
```

The first 10 rows of the resulting *AgeSum2b* data set are listed in ***Output 10***.



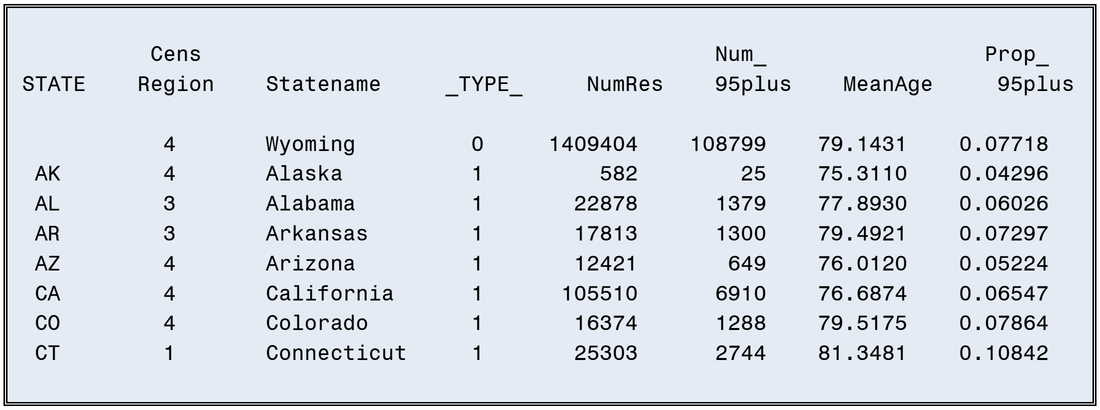
```
         Cens                                          Num_      Mean     Pct_
STATE    Region    Statename                 NumRes    95plus     Age    95plus

 RI        1       Rhode Island               8,221       928    83.01    11.29
 CT        1       Connecticut               25,303     2,744    81.35    10.84
 NH        1       New Hampshire              6,960       704    82.48    10.11
 MA        1       Massachusetts             43,152     4,297    81.29     9.96
 VT        1       Vermont                    2,804       271    82.17     9.66
 PA        1       Pennsylvania              81,267     7,086    81.13     8.72
 NY        1       New York                 109,754     9,510    79.57     8.66
 NJ        1       New Jersey                47,190     3,870    79.20     8.20
 ME        1       Maine                      6,408       504    81.73     7.86
 ND        2       North Dakota               5,661       746    83.74    13.18
```

Output 10. The first 10 rows of the PROC SQL summary data set with age measures aggregated by state and, ordered from highest to lowest percent of residents aged 95 and older within each census region.

There is one more variation I want to discuss before moving on to the next example task. Recall that we used the NWAY option with PROC SUMMARY so that the output data set includes only the rows for each state – that is, there is no overall row (i.e. the US average info). If we want an overall row, with PROC SUMMARY all we need to do is remove the NWAY option.

```
PROC SUMMARY DATA = Residents2012;
CLASS state ;
VAR age Age_ge95;
OUTPUT OUT = agesum3 (DROP = _FREQ_)
         N(age) = NumRes
     SUM(Age_ge95) = Num_95plus
        MEAN = MeanAge Prop_95plus ;
ID CensRegion StateName ;
RUN;
```

Well, sort of. The output may not be precisely what we expected. I kept the *_TYPE_* variable on the data set to help decipher what is happening. Take a look at ***Output 11***.



```
         Cens                                        Num_                  Prop_
STATE    Region    Statename     _TYPE_    NumRes    95plus    MeanAge     95plus

           4       Wyoming         0      1409404    108799    79.1431    0.07718
 AK        4       Alaska          1          582        25    75.3110    0.04296
 AL        3       Alabama         1        22878      1379    77.8930    0.06026
 AR        3       Arkansas        1        17813      1300    79.4921    0.07297
 AZ        4       Arizona         1        12421       649    76.0120    0.05224
 CA        4       California      1       105510      6910    76.6874    0.06547
 CO        4       Colorado        1        16374      1288    79.5175    0.07864
 CT        1       Connecticut     1        25303      2744    81.3481    0.10842
```

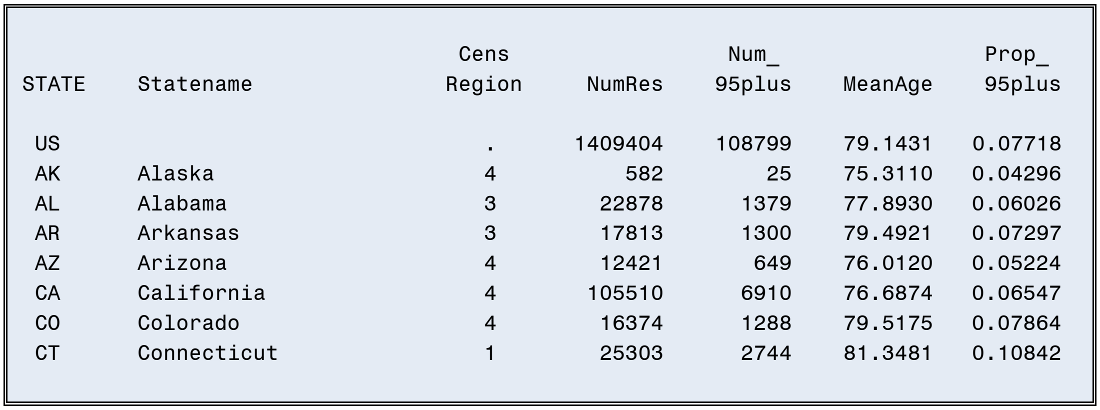
Output 11. The PROC SUMMARY result when we remove the NWAY option but retain ID variables is unexpected.

The overall summary row is the one with *_TYPE_* = 0. This is the US row. Unfortunately, State is missing because this is the way that PROC SUMMARY indicates that a given classification variable was not used in the aggregation for that row. This is made even more confusing because the values for

CensRegion and *StateName* for this row are NOT missing, and they are “wrong”. Recall what I said earlier about ID variables – here, because they are NOT populated from the input data for the overall row, they are assigned the values from the last row – which is Wyoming, since WY is alphabetically the last state abbreviation. We can get around this limitation by adding *CensRegion* and *Statename* to the CLASS statement along with judicious use of the TYPES statement, as shown below.

```
PROC FORMAT;
VALUE $msUS ' ' = 'US' ;
RUN;

PROC SUMMARY DATA = Residents2012 ;
CLASS CensRegion state StateName;
TYPES () CensRegion*State*StateName ;
VAR age Age_ge95;
OUTPUT OUT = agesum3a (DROP = _:)
         N(age) = NumRes
     SUM(Age_ge95) = Num_95plus
        MEAN = MeanAge Prop_95plus  ;
FORMAT state $msUS. ;
RUN;
```

For more about the very handy TYPES statement (and its cousin the WAYS statement), see the SAS documentation or look for user papers on the topic (e.g. Williams, 2006). This example also makes use of a “trick” which I can’t in good conscience recommend that you use unless you really know your data! I’m formatting the missing value of state on the output data set to print as ‘US’, and one could do something similar for *Statename* and *CensRegion*. This is probably OK to do in this case because there is no truly missing data on the CLASS variables (and even if there were PROC SUMMARY would by default exclude it since we are not using the MISSING option), but in general blithely formatting missing values to “look” like something else is probably not good practice. Likely a safer technique would be to use the _TYPE_ variable in a subsequent DATA step to assign desired values to *State, Census Region* and *StateName* for the overall row(s). A portion of the result is shown in ***Output 12***.



```
                                      Cens                 Num_                 Prop_
STATE    Statename                   Region     NumRes    95plus    MeanAge    95plus

 US                                     .      1409404    108799    79.1431   0.07718
 AK      Alaska                         4          582        25    75.3110   0.04296
 AL      Alabama                        3        22878      1379    77.8930   0.06026
 AR      Arkansas                       3        17813      1300    79.4921   0.07297
 AZ      Arizona                        4        12421       649    76.0120   0.05224
 CA      California                     4       105510      6910    76.6874   0.06547
 CO      Colorado                       4        16374      1288    79.5175   0.07864
 CT      Connecticut                    1        25303      2744    81.3481   0.10842
```

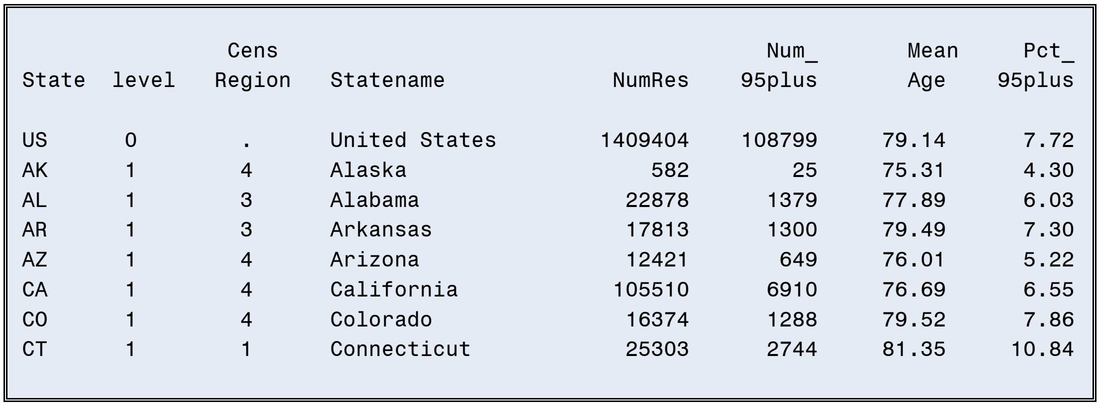
Output 12. PROC SUMMARY output for the US and each state, along with State name and Census region, courtesy of the TYPES statement.

So, how do we do the same thing with PROC SQL? It is a little tricky, but it demonstrates some additional SQL syntax. The code is shown below, followed by some explanation.

```
PROC SQL ;
CREATE TABLE agesum2d AS
(SELECT 0 AS level
       ,. AS CensRegion
       ,'US' AS State
        ,'United States' AS Statename
        ,N(age) AS NumRes
        ,SUM(Age_ge95) AS Num_95plus
        ,MEAN(Age) AS MeanAge FORMAT=6.2
        ,100*MEAN(Age_ge95) AS Pct_95plus FORMAT=6.2
FROM Residents2012)
UNION CORRESPONDING
(SELECT 1 AS level
     ,CensRegion
     ,State
     ,Statename
     ,N(age) AS NumRes
     ,SUM(Age_ge95) AS Num_95plus
     ,MEAN(Age) AS MeanAge FORMAT=6.2
     ,100*MEAN(Age_ge95) AS Pct_95plus FORMAT=6.2
FROM Residents2012
GROUP BY CensRegion, State, Statename )
ORDER BY level, State   ;
QUIT;
```

The UNION set operator allows us to stack or concatenate the results of two queries or SELECT clauses. The CORRESPONDING option is important to make sure that like columns align. The first SELECT clause gets the overall (US) statistics (Note the absence of a GROUP BY clause for the first SELECT), and as a stand-alone would produce a single row. The second query is much like we saw earlier on, where we include census region, state and state name on the GROUP BY, and will generate a row for each state (plus DC). Each of the SELECT clauses (which could each stand alone) is placed in parentheses. Outside the parentheses is the backbone of the overall CREATE TABLE query, which includes CREATE TABLE…, the set operator (UNION) and the final ORDER BY, which operates after the two SELECT results have been stacked.

The first several rows of the data set produced by this query are shown in ***Output 13***. There are a few other features of the code to note. First, it is critical to assign values to *CensRegion*, State and *StateName* in the first SELECT. If these columns were SELECTed from the source data, it would cause a Re-merge, which we definitely do not want. On the other hand if they were left off the first query, they would get dropped entirely from the resulting data set because of the way that UNION and CORRESPONDING work (Williams, 2012). Second, I create the column “LEVEL”, which functions sort of like *_TYPE_* from PROC SUMMARY in order to be able to put the resulting table in the desired order – with the US row first, and then the rest of the states in alphabetic order. Without this (if the ORDER BY included only *State*), the US row would come out between Texas (TX) and Utah (UT)!



```
                 Cens                                     Num_       Mean     Pct_
State   level   Region   Statename              NumRes   95plus       Age    95plus

US        0       .      United States         1409404   108799     79.14      7.72
AK        1       4      Alaska                    582       25     75.31      4.30
AL        1       3      Alabama                 22878     1379     77.89      6.03
AR        1       3      Arkansas                17813     1300     79.49      7.30
AZ        1       4      Arizona                 12421      649     76.01      5.22
CA        1       4      California             105510     6910     76.69      6.55
CO        1       4      Colorado                16374     1288     79.52      7.86
CT        1       1      Connecticut             25303     2744     81.35     10.84
```

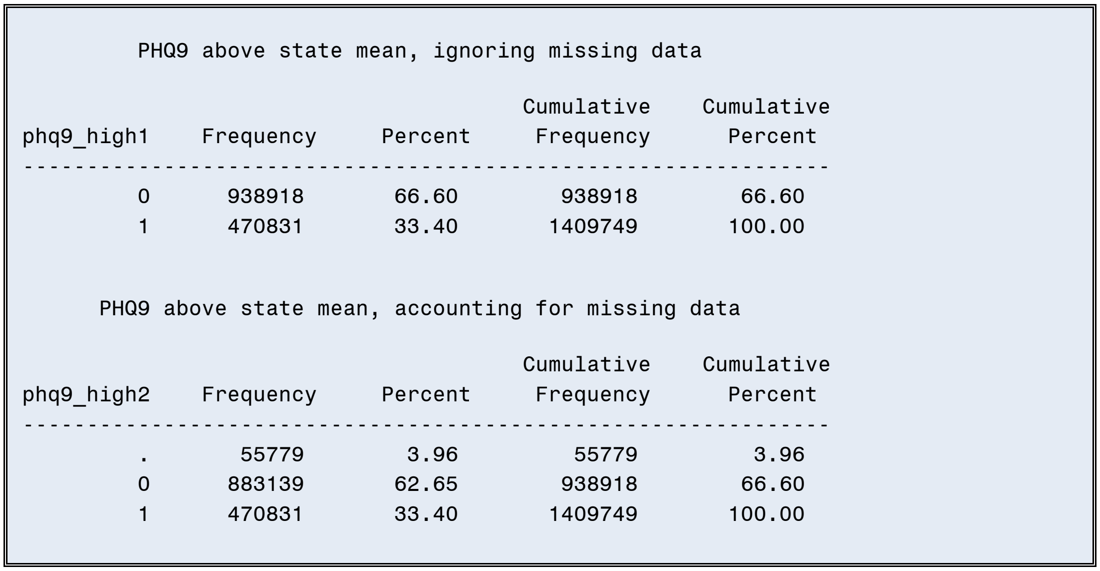
Output 13. This example illustrates using PROC SQL to generate statistics at two different levels of aggregation (overall US and by state)

EXAMPLE 3: COMBINE AGGREGATED VALUES WITH INDIVIDUAL VALUES

Sometimes you need to have the individual value joined with the aggregate value on the same observation. For example, you might want to identify values that are above the mean for the state. With PROC SUMMARY, we need to do this in two steps – one to generate the summary-level values and one to merge these back in with the individual-level data. For this example, the measure is the PHQ9, which is a brief ascertainment of resident mood, with higher values possibly indicative of depression.

```
*Determine if resident is above or below the median PHQ9 score for his/her state;
PROC SUMMARY DATA = Residents2012 NWAY;
CLASS state ;
VAR phq9 ;
OUTPUT OUT=PHQsum1 (DROP= _:) MEAN=StAvg_phq9 ;
RUN;

DATA phqsum1a ;
 MERGE Residents2012
       PHQsum1 ;
BY state ;
phq9_high1 = (phq9 > StAvg_phq9);
IF phq9 NE . then phq9_high2 = (phq9 > StAvg_phq9);

ATTRIB
 phq9_high1 LENGTH=3 LABEL='PHQ9 above state mean, ignoring missing data'
 phq9_high2 LENGTH=3 LABEL='PHQ9 above state mean, accounting for missing data' ;
RUN;
```

There is some missing data on the PHQ9; so, for illustration purposes, I create two different indicators of having a high (above the state mean) score: *PHQ9_High1* will be 0 if the resident has a missing PHQ9 value, and *PHQ9_High2* will be missing if PHQ9 is missing. I think the correct choice depends on the purpose, but I show both for two reasons – first, to make a point that the programmer should make a conscious choice about how to deal with missing data (rather than SAS ‘deciding’) and second, to illustrate ways to make the distinction in the DATA step and with PROC SQL. (We’ll deal later with missing data for the classification variable). For illustration, ***Output 14*** shows a PROC FREQ result for both the indicators.



```
         PHQ9 above state mean, ignoring missing data

                                    Cumulative    Cumulative
phq9_high1    Frequency     Percent     Frequency      Percent
---------------------------------------------------------------
         0      938918       66.60        938918        66.60
         1      470831       33.40       1409749       100.00


      PHQ9 above state mean, accounting for missing data

                                    Cumulative    Cumulative
phq9_high2    Frequency     Percent     Frequency      Percent
---------------------------------------------------------------
         .       55779        3.96         55779         3.96
         0      883139       62.65        938918        66.60
         1      470831       33.40       1409749       100.00
```

Output 14. Frequency of two indicators of high (above the state median) PHQ9 mood scores, illustrating two different ways to handle missing data

With SQL, you can take advantage of the “re-merge” and get the summary and individual data on the same row with a single step. And the ‘SELECT *’ syntax will pull ALL of the columns from the original resident-level file – to which we add the two high-PHQ9 indicators (commas separating the columns, of course!). On the other hand, getting SQL to account for missing data – that is, creating the high-PHQ9 indicator so that it will have a missing value if the individual PHQ9 value is missing – is a little more complicated in SQL, requiring the use of the CASE expression, the syntax of which may take some getting used to for DATA Step die-hards! But it works, and the result is identical to what we obtained with the combination of PROC SUMMARY and the DATA Step merge. I don’t repeat the output, but will point out that we get a NOTE in the log, informing us of the re-merge (refer to ***Output 8***), but in this case, we intended the re-merge, so we can safely ignore this note.

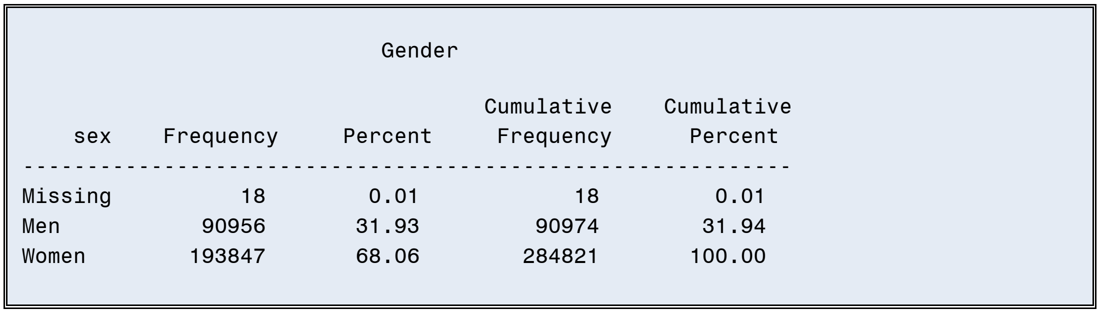
```
PROC SQL ;
CREATE TABLE phqsum2 as
SELECT *
     ,MEAN(phq9) as StAvg_phq9 FORMAT=6.1
     ,(phq9 > CALCULATED StAvg_phq9) AS phq9_high1
        LENGTH=3 LABEL='PHQ9 above state mean, ignoring missing data'
     ,CASE
        WHEN (phq9 > CALCULATED StAvg_phq9) THEN 1
       WHEN (0 LE phq9 le CALCULATED StAvg_phq9) THEN 0
      ELSE .
       END AS phq9_high2
       LENGTH=3 LABEL = 'PHQ9 above state mean, accounting for missing data'

FROM Residents2012
GROUP BY state
ORDER BY state, provnum, ResID ;
QUIT;
```

To take this example a little further, let’s say we don’t want to just add these indicators to our file, but we want to select based on them. And, we add just a little twist. Let’s say what we want is to select those who are two or more points above the gender-specific state average on the PHQ – so, now we have two classification variables (*STATE* and *Sex*). Note that there a small number of observations are missing *Sex*, so again we need to be conscious of how we deal with these. First, let’s include those missing *Sex* as a third category. To do this with PROC SUMMARY we add the MISSING option to the PROC

SUMMARY statement. This has no effect on the treatment of the analysis variable – missing values on *PHQ9* do not enter in to the calculation of the state average. We use NWAY so that we get only the "two-way" rows, that is, no overall state rows, overall sex rows or the grand mean. So, the data set that we output from PROC SUMMARY will have one observation per state-sex combination and for every state that has any residents with missing *Sex*, an additional observation will be output. We do have to sort our original data set (which is rather large) by *STATE* and *Sex* in order to merge the state-gender averages back in. And then we output all the rows that are more than 2 points above the state-gender mean (ignoring missing data on *PHQ9*).

```
PROC SUMMARY DATA = Residents2012 NWAY MISSING ;
CLASS state sex;
VAR phq9 ;
OUTPUT OUT=PHQsum_bySex1 (DROP= _:) MEAN=StAvg_phq9_bySex ;
RUN;

PROC SORT DATA = Residents2012 ;
BY state sex ;
RUN;

DATA PHQ_High1 ;
 MERGE Residents2012 PHQsum_bySex1 ;
BY state sex ;

phq9_high1 = (phq9 > (StAvg_phq9_bySex + 2));
IF phq9_high1;
RUN;
```

The resulting data set is resident-level, including only the individuals with high PHQ9 scores, and the frequency distribution by gender is shown in ***Output 15***. (As an aside, clearly women greatly outnumber men among those with high PHQ9 scores, but they also greatly outnumber men in the nursing home population as a whole – about 2 to 1).



```
                              Gender

                                          Cumulative    Cumulative
    sex    Frequency     Percent     Frequency      Percent
------------------------------------------------------------
Missing          18        0.01            18         0.01
Men           90956       31.93         90974        31.94
Women        193847       68.06        284821       100.00
```

Output 15. Gender distribution of those with high PHQ9 scores, defined as more than 2 points above the gender-state specific mean

The PROC SQL code to achieve the same result is below:

```
PROC SQL ;
CREATE TABLE PHQ_high2 as
SELECT *
     ,MEAN(phq9) as StAvg_phq9_bySex FORMAT=6.1
     ,(phq9 > (CALCULATED StAvg_phq9_bySex + 2)) AS phq9_high1
FROM Residents2012
GROUP BY state, sex
HAVING phq9_high1 = 1
ORDER BY state, provnum, sex, ResID ;
QUIT;
```

The GROUP BY clause dictates the groupings for which the summary functions (here the MEAN) are calculated, and SQL treats missing values on the GROUP BY variables just the same as non-missing values (so we get the same result as for PROC SUMMARY with the MISSING option. Because we are selecting columns other than the GROUP BY columns and the summary functions, SQL does a re-merge. The CALCULATED keyword before StAvg_phq9_bySex is required because without it SQL expects all columns referred to in the SELECT expression to be on the table(s) referred to in the FROM clause. The HAVING expression acts at the GROUP level and you must use it (rather than WHERE) when the selection is based on a summary function. The PHQ_high2 data set created by the PROC SQL code is identical to the PHQ_high1 data set created by PROC SUMMARY, PROC SORT and the DATA step.

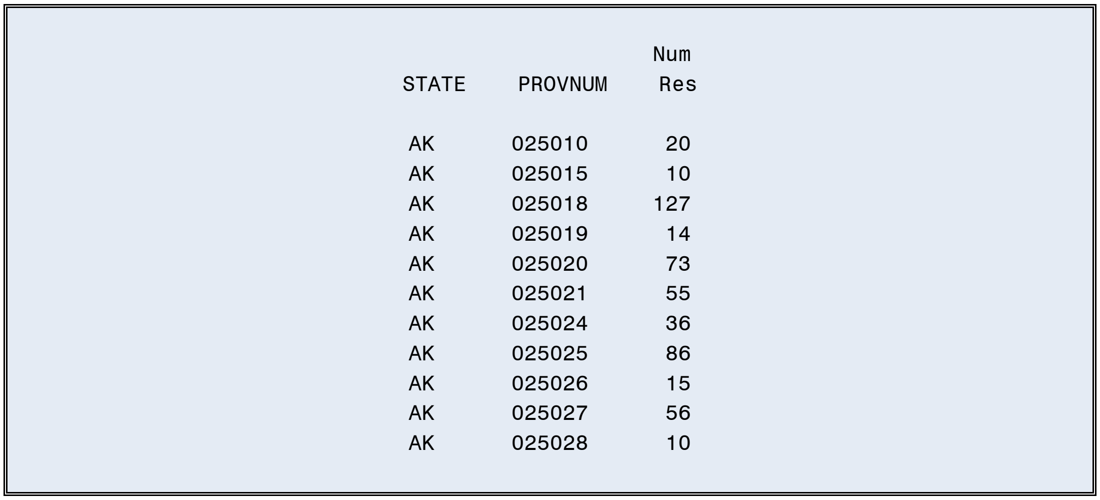
I make one final note before moving on to the next example. If you did not want to include those with missing SEX in the summary calculations or in identifying those with high values, here is what to do: With the PROC SUMMARY method, remove the MISSING option from the PROC SUMMARY statement, and then, in the DATA step add the stipulation that "*Sex* NE ." to the subsetting IF statement (or to the definition of *PHQ_high*). This is important because those with missing *Sex* will be missing *StAvg_phq9_bySex*, and thus, assuming they had a non-missing PHQ9 score would ALWAYS get selected since a non-missing value is always greater than a missing value. With PROC SQL, the simplest thing to do would be to add a WHERE clause (it must be directly after the FROM clause) stating "WHERE *Sex* NE .".

EXAMPLE 4: NESTING SUMMARY FUNCTIONS

This next example may at first seem quite similar to the previous ones, but as we go through it, you'll see that it requires somewhat different logic. Here we want to select the nursing home with the largest number of residents in each state. The nursing home identifier is called *ProvNum*. Thinking about PROC SUMMARY, we can use it to count the number of residents in each nursing home, but a separate step will be required to identify the largest number (or MAX) of residents – essentially we need to perform a summary function (identifying the max by state) on the summary function (count of residents per provider). Here is the code for the PROC SUMMARY method:

```
PROC SUMMARY DATA = Residents2012 NWAY;
CLASS state provnum ;
VAR ResID ;
OUTPUT OUT=ResPerFac1 (DROP = _:) N=NumRes ;
RUN;

PROC SUMMARY DATA = ResPerFac1 NWAY ;
CLASS state ;
VAR NumRes ;
OUTPUT OUT=MaxResPerState (DROP = _:)
       MAX=MaxRes ;
RUN;

* merge the MaxPerState back with the provider-level summary;
DATA MostRes1 ;
MERGE ResPerFac1
      MaxResPerState;
BY state ;
IF NumRes = MaxRes ;
RUN;
```

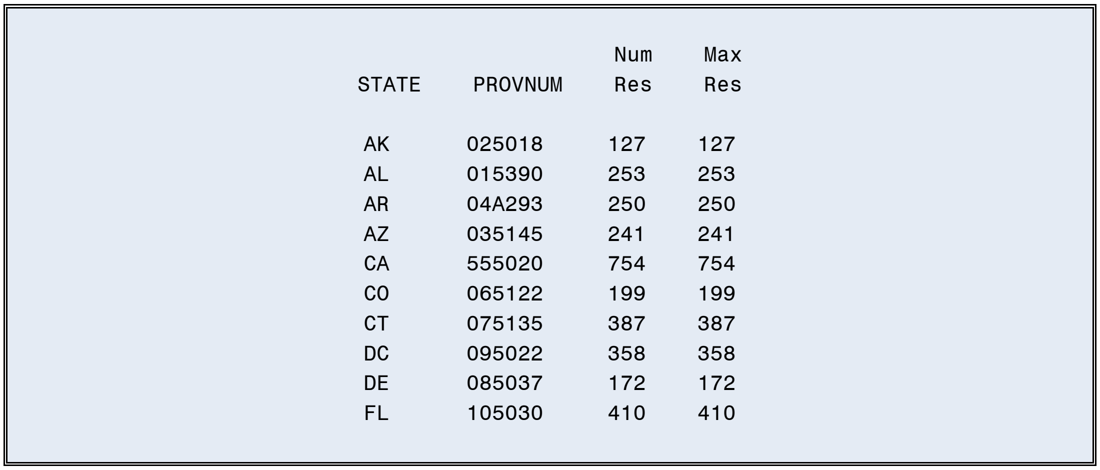
In the first PROC SUMMARY, one row is output for each nursing home (provnum). I put state on the CLASS statement so that it would also be on the resulting file (and the data will be sorted by state and provnum). Just to give a concrete picture of the result of this first step, the first 10 observations are shown in ***Output 16***.



```
                         Num
STATE     PROVNUM        Res

 AK        025010         20
 AK        025015         10
 AK        025018        127
 AK        025019         14
 AK        025020         73
 AK        025021         55
 AK        025024         36
 AK        025025         86
 AK        025026         15
 AK        025027         56
 AK        025028         10
```

Output 16. Ten observations showing PROC SUMMARY result of counting the number of residents in each nursing home (PROVNUM)

The second PROC SUMMARY step takes the result of the first one as its input. Here, because we want to get the largest number of residents per state, we CLASS on state only and use the MAX function. We cannot identify the largest provider in this same step because the output is state-level. So a third step – a DATA Step MERGE is needed to combine the provider-level summary info with the state-level summary info, and make the selection. The result has the largest provider in each state. Note that if there were a "tie" for largest nursing home, all the providers with that largest value would be included. The result for the first 10 states is shown in ***Output 17***.



```
                         Num      Max
STATE     PROVNUM        Res      Res

 AK        025018        127      127
 AL        015390        253      253
 AR        04A293        250      250
 AZ        035145        241      241
 CA        555020        754      754
 CO        065122        199      199
 CT        075135        387      387
 DC        095022        358      358
 DE        085037        172      172
 FL        105030        410      410
```

Output 17. Ten observations showing PROC SUMMARY result identifying the nursing home in each state with the largest number of residents

The SQL code to achieve the same result appears a little complicated at first, but it is really quite elegant. SQL will not allow you to nest summary functions directly, but you can nest queries, which has the same effect.

Here is the code:

```
PROC SQL ;
CREATE TABLE MostRes2 AS
SELECT * FROM
    (SELECT state
      , provnum
      , COUNT(ResID) AS NumRes
    FROM Residents2012
    GROUP BY state, provnum)
GROUP BY state
HAVING NumRes = MAX(NumRes)
ORDER BY state;
QUIT;
```

Look first at the nested query in parentheses. This part does the same thing as the first PROC SUMMARY above – namely, calculates the number of residents in each nursing home. It creates a sort of virtual table, called an in-line view. That virtual table is then queried in the outer query (starting with SELECT *); this selects the rows from the in-line view (GROUPed BY state) that meet the criteria of having the *NumRes* (provider-level) equal to the maximum value, defined at the state level by virtue of the GROUP BY clause. This yields an identical result to the SUMMARY method.

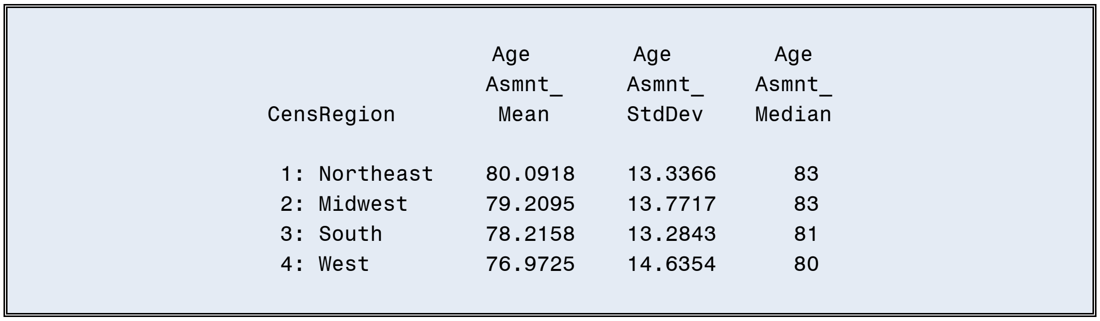
EXAMPLE 5: SUMMARY STATISTICS FOR A DERIVED COLUMN

The last example is a fairly simple one, but I include it to demonstrate another nice feature of PROC SQL – the capacity to compute summary functions on new (or CALCULATED) columns. Let's say we want to compute descriptive statistics by census region on resident age at the date of the assessment, and while we have date of birth and assessment date on our resident file, we don't have age at assessment computed. Unfortunately, we can't put an expression on the VAR statement in PROC SUMMARY, so we have to do this in two steps.

The code is shown below. In the DATA step we calculate resident age at assessment, based on birth date (*Dob*) and assessment date (*AsmntDate*), both SAS dates. There are many different ways to compute age, but I like this one, which I learned from Ron Cody (Cody 2004). And then in the second step (PROC SUMMARY) we simply calculate the desired summary statistics, with census region (*CensRegion*) as the CLASS variable.

```
DATA CalcAge ;
 SET Residents2012 ;

 AgeAsmnt = FLOOR(YRDIF(dob,AsmntDate,"ACTUAL" )) ;
 RUN;

PROC SUMMARY DATA = CalcAge NWAY;
CLASS CensRegion ;
VAR AgeAsmnt ;
OUTPUT OUT=SumCalcAge1 (DROP = _:)
      MEAN=
      STD=
      MEDIAN= / AUTONAME ;
RUN;
```

The full result (just four observations) is PRINTed in ***Output 18***.



```
                    Age        Age        Age
                    Asmnt_     Asmnt_     Asmnt_
CensRegion           Mean      StdDev     Median

 1: Northeast       80.0918    13.3366      83
 2: Midwest         79.2095    13.7717      83
 3: South           78.2158    13.2843      81
 4: West            76.9725    14.6354      80
```

Output 18. Summary statistics on resident age at assessment, by census region

A single PROC SQL step can get us the same result. The code is shown below. Note that if we wanted to round or format or otherwise transform the summary statistics, this could also be done in the same step, while with PROC SUMMARY, post-processing (probably another DATA Step) would be required. I will also point out that we don't actually SELECT the implied AgeAsmnt column – because that would result in an undesired re-merge, resulting in a data set with a row for each resident. So, if you wanted to have the age at assessment on the resident-level data set for future use, the DATA Step and PROC SUMMARY method might be preferable.

```
PROC SQL ;
CREATE TABLE SumCalcAge2 AS
SELECT CensRegion
     ,MEAN(FLOOR(YRDIF(dob,AsmntDate,"ACTUAL" ))) AS AgeAsmnt_Mean
     ,STD(FLOOR(YRDIF(dob,AsmntDate,"ACTUAL" ))) AS AgeAsmnt_SD
     ,MEDIAN(FLOOR(YRDIF(dob,AsmntDate,"ACTUAL" ))) AS AgeAsmnt_Md
FROM Residents2012
GROUP BY CensRegion ;
QUIT;
```

CONCLUSIONS

I confess that I still use PROC SUMMARY a lot. However, whenever I find myself following PROC SUMMARY with a MERGE back into my original file or needing to do other pre- or post-processing in conjunction with PROC SUMMARY, I ask myself whether PROC SQL couldn't get the job done with fewer steps. I will also admit that PROC SQL can seem like a bit of a "black box" and it does "think" differently than the DATA step and PROC SUMMARY, which means – as when you are developing ANY new code – you should check that you are getting the result you want[1]. I hope that the examples in this paper have taken a little of the mystery out of PROC SQL and demonstrated how useful it can be for a variety of data aggregation tasks. Nonetheless, PROC SUMMARY has some real advantages too, such as the AUTONAME and AUTOLABEL features, and, of course, it can compute quite a few statistics (such as different quantiles) that are not available in PROC SQL. SAS often does provide multiple paths to the same goal with data manipulations, but there are often subtle (or not so subtle) differences in the methods that may make one a better choice than another in a particular situation. Part of being a truly accomplished SAS programmer is knowing the different methods in sufficient detail that you can choose the right tool for the task. Best of Luck!

[1] As a side note, one technique I use frequently when testing new code is to use PROC COMPARE to check that an old method and a new method give the same result. In fact, I used this very handy procedure to check all the examples in this paper – that is, to check that the PROC SUMMARY and PROC SQL methods really did give identical results.

REFERENCES AND RECOMMENDED READING

Centers for Medicare and Medicaid Services, “Nursing Home Data Compendium 2013 Edition”, Available at *https://www.cms.gov/Medicare/Provider-Enrollment-and-Certification/CertificationandComplianc/downloads/nursinghomedatacompendium_508.pdf*

Cody, Ron. 2010. *SAS Functions by Example, 2nd edition*. Cary, NC. SAS Institute Inc.

Schreier, Howard. 2008. *PROC SQL by Example: Using SQL within SAS®.* Cary, NC: SAS Institute Inc.

Williams, C. 2012. “Queries, Joins and WHERE clauses, Oh My! Demystifying PROC SQL”. *Proceedings of SAS Global Forum 2012*. Available at: *http://support.sas.com/resources/papers/proceedings12/149-2012.pdf*

Williams, Christianna. 2006. “Any WAY you want it: Getting the right TYPEs of Observations out of PROC SUMMARY or MEANS” *Proceedings of NESUG 2006.* Available at: *http://www.lexjansen.com/nesug/nesug06/cc/cc30.pdf*

Williams, Christianna. 2012. “PROC SQL for DATA Step Die-Hards” *Proceedings of NESUG 2012.* Available at: *http://www.lexjansen.com/nesug/nesug12/hw/hw03.pdf*

CONTACT INFORMATION

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

Name: Christianna Williams
E-mail: Christianna.S.Williams@gmail.com