Paper 6481-2016

Mastering Data Summarization with PROC SQL

Christianna Williams PhD, Chapel Hill, NC

ABSTRACT

The SQL procedure is extremely powerful when it comes to summarizing and aggregating data, but it can be a little daunting for programmers who are new to SAS® or for more experienced programmers who are more familiar with using the SUMMARY or MEANS procedure for aggregating data. This hands-on workshop demonstrates how to use PROC SQL for a variety of summarization and aggregation tasks. These tasks include summarizing multiple measures for groupings of interest, combining summary and detail information (via several techniques), nesting summary functions by using inline views, and generating summary statistics for derived or calculated variables. Attendees will learn how to use a variety of PROC SQL summary functions, how to effectively use WHERE and HAVING clauses in constructing queries, and how to exploit the PROC SQL remerge. The examples become progressively more complex over the course of the workshop as you gain mastery of using PROC SQL for summarizing data.

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 purpose of this paper and the accompanying exercises is to demonstrate how to achieve various summarization and aggregation tasks using PROC SQL. For readers who are more familiar with using PROC SUMMARY/PROC MEANS for these types of tasks, I refer you to an earlier paper that described how to perform many of the same manipulations with both methods (Williams, 2015).

I wanted to make the exercises 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 2014. These data (in many different summary forms, thanks to PROC SQL and PROC SUMMARY!) are reported in the 2015 edition of the Centers for Medicare and Medicaid Nursing Home Data Compendium (CMS, 2016).

EXERCISE 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 helpful features of PROC SQL. On my data file the variable DOB is the resident's birth date, and it is a SAS date formatted as YYMMDD8. (i.e. yy-mm-dd). SAS dates, of course, are integers, with values reflecting 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. First, I just show the birthdates of the first 5 residents in the file (**Output 1)**.

```
Print Date of Birth for 5 Residents

  Obs      dob      age

  1    29-06-02       85
  2    27-11-24       87
  3    30-01-18       85
  4    21-05-24       93
  5    40-02-17       75
```

Output 1. Dates of birth and age for the first 5 observations in our sample file. Date of birth is in the FORMAT YY-MM-DD.

The first exercise (Exercise 1.1) is simply to produce selected summary statistics on date of birth. Here is the code

```
TITLE3 'Exercise 1.1 Generate various summary statistics for date of
  birth';
PROC SQL ;
SELECT N(dob) AS dob_N
      ,MIN(dob) AS dob_min
      ,MAX(dob) AS dob_max
    ,MEDIAN(dob) AS dob_med
FROM in.ressamp2014;
QUIT;
```

Because I did not begin the query with a CREATE TABLE clause, the result will go to the active output destination, and it is shown in **Output 2**.

Exercise 1: Aggregate a single measure over the entire file
1.1 Generate various summary statistics for date of birth

dob_N	dob_min	dob_max	dob_med
351570	-21914	20080	-9917

Output 2. Output from Exercise 1.1, summary statistics on date of birth for the resident sample

There are a few things to note about this output. The most obvious is that, although the source variable DOB had a date FORMAT, the summary statistics do not. We will rectify that in the next part of this exercise. Second, notice that in the code each column was assigned a column (or variable name). This is important. You can try on your own to determine what will happen if the "as dob_N" or other as xxx expressions are left off. No error message is generated; however, in the output, there would be no heading(s) on the column(s) that had not been assigned an alias. And, if you had been creating a new data set containing the summary statistics, as we will in many of the later exercises, SAS would assign variable name(s) to those columns for which you had not provided one – as all SAS variables must have names – and I can guarantee you that you would not be happy with the names that SAS came up with – try it! Finally, just a few comments about the way these summary functions work. In general, they act only on non-missing values, so the N is the count of the rows with non-missing dates of birth, and the min, max and median are also just for the non-missing values. And one last note about the MEDIAN function – this very useful function is available in PROC SQL only as of SAS 9.4. With prior versions of SAS, you will not get an error, but you will cause an unintended re-merge and not get the desired result – you'll have to use PROC SUMMARY or PROC MEANS.

Ok. For the next part of this exercise, let's assign some attributes to our summary columns, including FORMATs and LABELs. For fun, I'm playing around with the wide selection of date FORMATs that SAS provides, and for variety, I'm also demonstrating a few other summary functions:

```
TITLE3 'Exercise 1.2:  Add some attributes - including FORMATs';
PROC SQL ;
CREATE TABLE dobstats2 AS
SELECT N(dob) AS dob_N LABEL='DOB count' FORMAT=COMMA10.
    ,NMISS(dob) AS dob_NMiss LABEL='DOB # missing' FORMAT=COMMA4.
      ,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=WORDDATE17.
    ,RANGE(dob) AS dob_range LABEL='Days between first and last DOB'
  FORMAT=best12.
FROM in.ressamp2014;
QUIT;
```

The result is shown in **Output 3**.

Exercise 1: Aggregate a single measure over the entire file
Exercise 1.2: Add some attributes - including FORMATs

DOB count	DOB # missing	Earliest DOB	Latest DOB	Median DOB	Days between first and last DOB
351,570	0	01/01/1900	23DEC2014	Nov 6, 1932	41994

Output 3. Output for Exercise 1.2 – summary statistics for date of birth after addition of formats and labels

The last part of this first exercise mostly falls into the "because we can" department – just showing off some other FORMATs and SAS functions. Let's say we want to determine the day of the week of the birthday of the oldest resident and the number of years between the oldest and most recent birth date. See below:

```
TITLE3 'Exercise 1.3. Fun with functions and FORMATs';
TITLE4 'Determine day of week of oldest birthday' ;
TITLE5 'And number of years between oldest and most recent';
PROC SQL ;
CREATE TABLE dobstats3 AS
SELECT
   MIN(dob) AS dob_Min LABEL = 'Earliest DOB' FORMAT=downame10.
  ,RANGE(dob) AS dob_range
            LABEL='Days between first and last DOB' FORMAT=best12.
  ,yrdif(CALCULATED dob_Min),max(dob),'Actual') as range_yrs
            LABEL='years between first and last dob (yrdif)'
  ,intck('YEAR',CALCULATED dob_min,max(dob)) as range_yrs2
            LABEL='years between first and last dob (intck)'
FROM in.ressamp2014;
QUIT;
```

Did you know about the DOWNAME (i.e., day of week name) FORMAT?. The things we learn from just browsing the documentation! I'm also showing two different ways to compute the number of years between two SAS dates, with the YRDIF and INTCK functions. There really is something to learn here about PROC SQL also. Although the first summary column we generate assigns the column name dob_min to the earliest date of birth, we will get an error if we do not use the CALCULATED keyword when we refer to dob_min later in the query. The error message will say "*ERROR: The following columns were not found in the contributing tables: dob_min*."

The result, a data set named DOBSTATS3 with one observation and 4 variables, is shown in **Output 4**. Note that although the earliest date of birth just prints as Monday, this is only because of the FORMAT we used: the underlying value is exactly what it was in Exercise 1.1, that is, -21914, which corresponds to January 1, 1900. Another thing to note is the differing results of the YRDIF and INTCK functions. YRDIF calculates the number of years between two dates, as a real number; hence, unless we round it, it will not show up as an integer. In contrast, the INTCK function counts the number of interval boundaries (here year boundaries – that is the number of January 1sts) between two dates, and this is, a count, and thus an integer. The difference could matter for some applications – know your functions!

Exercise 1: Aggregate a single measure over the entire file
1.3. Fun with functions and FORMATs
Determine day of week of oldest birthday
And number of years between oldest and most recent

Earliest DOB	Days between first and last DOB	years between first and last dob (yrdif)	years between first and last dob (intck)
Monday	41994	114.975	114

Output 4. Output for Exercise 1.3 – summary statistics for date of birth after addition of formats and labels

EXERCISE 2: AGGREGATING MORE THAN ONE MEASURE WITH GROUPING

Shortly we will move on to obtaining summaries for specified subgroups, but first we just demonstrate that multiple variables can be aggregated within the same step, across the entire file. For Exercise 2.1, we compute descriptive statistics for resident age and PHQ, which is a measure of depressive symptoms and demonstrate the difference between the N function and the wild card COUNT(*) syntax.

```
TITLE3 'Ex 2.1 - Descriptive statistics on age and Depression ';
PROC SQL ;
CREATE TABLE agesum1 AS
SELECT N(Age) AS NumRes_withAge FORMAT=comma9.
     ,MEAN(Age) AS MeanAge
     ,STD(Age) AS stdAge
     ,N(PHQ9) AS NumRes_withPHQ FORMAT=comma9.
     ,Mean(PHQ9) AS MeanPHQ9
     ,STD(PHQ9) AS stdPHQ9
     ,COUNT(*) AS NumRows FORMAT=comma9.
FROM in.ressamp2014;
QUIT;
```

Note that the N function (as in N(Age and N(PHQ9) above provide counts of non-missing values for the variable that is their argument (just like the N keyword for PROC SUMMARY/MEANS). The COUNT() function (with a column name as its argument does exactly the same thing. In contrast, the COUNT(*) syntax counts the number of rows being aggregated, without regard to missing values on any or all columns. In this case we are aggregating across the entire data set RESSAMP2014, so COUNT(*) will return the number of observations in the data set. It is thus comparable to what you would see in the automatic variable _FREQ_ in output from PROC SUMMARY. See **Output 5**.

Exercise 2.1 - Descriptive statistics on age and Depression Score - full sample

NumRes_ withAge	MeanAge	stdAge	NumRes_ withPHQ	Mean PHQ9	stdPHQ9	NumRows
351,568	78.8493	13.7197	335,813	2.51738	3.63832	351,570

Output 5. Output for Exercise 2.1 – summary statistics for age and PHQ9

A handy extension of this, which we might want to do for a number of analytic variables is the ability to compute the percentage of rows with missing data on the PHQ9 – again across the entire sample. To do this with PROC SUMMARY and/or MEANS would require post-processing in a DATA step. Here is the code.

```
TITLE3 'Ex 2.2 – Calculate percent with missing data on PHQ9 ';

CREATE TABLE agesum2 AS
SELECT NMISS(phq9) AS N_missingPHQ9 FORMAT=COMMA9.
     ,COUNT(*) as NumRows FORMAT = COMMA9.
     ,(NMISS(phq9)/COUNT(*)) AS pct_missingPHQ9 FORMAT = PERCENT7.1
FROM in.ressamp2014;
QUIT;
```

As noted above, we could substitute "CALCULATED N_missingPHQ9" for "NMISS(phq9) and/or "CALCULATED NumRows" for "COUNT(*)" in the expression computeing pct_missingPHQ9. This can be helpful if the expression for the derived column(s) are long or complex. **Output 6** shows the result.

Exercise 2.2 - Calculate Percent with missing data

N_ missing PHQ9	NumRows	pct_ missingPHQ9
15,757	351,570	4.5%

Output 6. Output for Exercise 2.2 – Calculation of percent missing for an analytic variable

For the next exercise, we add a grouping variable. Specifically, we are computing summary statistics for resident age for each US state. In addition to integer Age, the RESSAMP2014 data set also has a variable named AGE_GE95, which is a 0,1 indicator of whether the resident is aged 95 years or older (i.e. it has a value of 1 if the resident is 95+ and 0 if he/she is less than 95). A useful feature of 0/1 "Boolean" variables like this is that one can easily get counts and proportions by using the SUM and Mean statistics, respectively, as we do for Exercise 2.3.

```
TITLE3 'Exercise 2.3 - Getting Age Summaries by State';
PROC SQL ;
CREATE TABLE agesum3 AS
SELECT state
      ,COUNT(*) AS NumRes
     ,MEAN(Age) AS MeanAge
     ,SUM(Age_ge95) AS Num_95plus
     ,MEAN(Age_ge95) AS Prop_95plus
FROM in.ressamp2014
GROUP BY state;
QUIT;
```

The GROUP BY syntax, which is required to come after the FROM clause specifies that the summary features should act on groups of rows with common values for the GROUP BY column(s) – here, state, and it is not necessary that the data be sorted on the grouping variable(s). COUNT(*) will give us the number of rows for each state (regardless of missing data); MEAN(Age) will give the average resident age for each state; and SUM(AGE_ge95) and MEAN(Age_ge95) will give us the number and proportion of residents in each state who are 95 or older. An important feature of this code is that no other columns are selected by the query besides the group by variable and summary statistics. We will come back to this point a little later.

Instead of just a single row as we got when summarizing across the entire file, the summary data set here has 51 rows – one for each state and the District of Columbia. The first few rows are shown in **Output 7**.

Exercise 2: Aggregating multiple measures and adding a grouping variable
Exercise 2.3 - Getting Age Summaries by State

STATE	Num Res	MeanAge	Num_ 95plus	Prop_ 95plus
AK	157	75.9745	5	0.03185
AL	5809	77.7297	336	0.05784
AR	4525	79.3017	332	0.07337
AZ	3123	74.8668	176	0.05636
CA	26631	76.5290	1770	0.06647
CO	4185	79.0927	326	0.07790
CT	6199	81.1284	683	0.11018
DC	662	75.9079	52	0.07855
DE	1098	79.6011	89	0.08106
FL	19247	79.3325	1487	0.07726

Output 7. Partial output for Exercise 2.3, state summaries for age in years and residents aged 95 or older.

As noted, there is no requirement with a GROUP BY clause that the data set being summarized is sorted. However, you may wish to control the order of the rows in the resulting summary data set. In the previous exercise, because the input data set happens to be sorted alphabetically by state postal abbreviation for state, the summary file retains this order. However, let's say we want to find out which states have the highest proportion of residents age 95 and older. We can get the list ordered in this way by simply adding an ORDER BY CLAUSE. Again, with PROC SQL, the order of the clauses is important – ORDER BY, if included, has to be after GROUP BY.

Since we want the states with the highest proportion of 95 year-olds to be first, we want to put the output in descending order. Unlike PROC SORT (or many other places in SAS), in PROC SQL, the keyword DESCENDING (which can be shortened to DESC), must be after the column name that it modifies. Also, even though the column PROP_95plus is calculated as part of this query (i.e. it does not exist on the input data set), the CALCULATED keyword is not required. The reason for this (as suggested by ORDER BY being the last clause in the query, the ordering of the rows does not happen until all of the columns have been computed; at this point PROP_95plus already exists.

```
TITLE3 'Exercise 2.4 - Getting Age Summaries by State';
TITLE4 'Put in descending order of proportion 95+';
PROC SQL ;
CREATE TABLE agesum4 AS
SELECT state
      ,COUNT(*) AS NumRes
    ,MEAN(Age) AS MeanAge
    ,SUM(Age_ge95) AS Num_95plus
    ,MEAN(Age_ge95) AS Prop_95plus
FROM in.ressamp2014
GROUP BY state
ORDER BY Prop_95plus DESCENDING ;
QUIT;
```

The first 10 rows of the result are shown in **Output 8**.

Exercise 2.4 - Getting Age Summaries by State
Put in descending order of proportion 95+

STATE	Num Res	MeanAge	Num_ 95plus	Prop_ 95plus
ND	1383	83.6602	181	0.13087
SD	1584	83.2898	200	0.12626
HI	927	80.9266	116	0.12513
RI	2054	82.9352	247	0.12025
MN	6714	81.9988	791	0.11781
IA	6228	82.4748	719	0.11545
WI	6878	82.0769	790	0.11486
CT	6199	81.1284	683	0.11018
VT	682	82.0630	75	0.10997
NH	1719	81.7213	181	0.10529

Output 8. Partial output for Exercise 2.4, state summaries for Age in years and residents aged 95 or older, with output ordered in descending order of the proportion aged 95+.

You are not limited to a single grouping variable. With multiple grouping variables (separated by commas in the GROUP BY clause), the summary statistics will be calculated for all groups based on the cross-classification of the group by variables. For example, if we used gender and state, the result would have 102 rows – 51 for women and 51 for men. In Exercise 2.5, we have the special case where one grouping variable (here STATE) is nested within another grouping variable (Census region). That is, each state is assigned to one and only one of 4 large geographic census regions. This is an easy way to add another variable to the summary dataset in the case where you have nested categories.

```
TITLE3 'Exercise 2.5 - Getting Age Summaries by State';
TITLE4 'Adding Census Region';
PROC SQL ;
CREATE TABLE AgeSum5 AS
SELECT state
      ,CensRegion
      ,N(age) AS NumRes
    ,SUM(Age_ge95) AS Num_95plus
    ,MEAN(Age) AS MeanAge
    ,MEAN(Age_ge95) AS Prop_95plus
FROM in.ressamp2014
GROUP BY state, censregion
ORDER BY censregion, state, Prop_95plus DESCENDING;
QUIT;
```

The first few rows of the result are shown in **Output 9**.

Exercise 2.5 - Getting Age Summaries by State
Adding Census Region

Cens Region	STATE	Num Res	Num_ 95plus	MeanAge	Prop_ 95plus
1	CT	6199	683	81.1284	0.11018
1	MA	10616	1077	81.0309	0.10145
1	ME	1586	121	81.5422	0.07629
1	NH	1719	181	81.7213	0.10529
1	NJ	11592	1032	78.8908	0.08903
1	NY	27072	2385	79.3003	0.08810
1	PA	20083	1850	80.9177	0.09212
1	RI	2054	247	82.9352	0.12025
1	VT	682	75	82.0630	0.10997
2	IA	6228	719	82.4748	0.11545

Output 9. Partial output for Exercise 2.5, grouping on census region and state, which is nested within census region

Very often analytic files have a natural clustering or grouping. In a longitudinal study where patients are followed over time, that grouping might be a patient identifier. Or it might be study site. In the nursing home data, each resident lives within a particular nursing home, and very often we are interested in characterizing the nursing home by summarizing across its residents. The next few exercises do this. So, in exercise 2.6, instead of computing summaries grouped by STATE, we group on facility – the federal provider number for each nursing home is called PROVNUM. We'll get to more complicated examples of this type of grouping in the next set of exercises; however, here the example is very similar to the previous except we group on PROVNUM.

```
TITLE3 'Exercise 2.6 - Facility summary of residents age 95+';
PROC SQL ;
CREATE TABLE AGESUM_PROVIDER AS
SELECT
    provnum
  ,N(AGE) AS NumRes
  ,SUM(Age_GE95) AS Num_95plus
  ,Mean(Age_GE95) AS Prop_95plus
FROM in.ressamp2014
GROUP  BY PROVNUM ;
QUIT;
```

Note that because we are selecting only the grouping variable and summary statistics, the resulting data set will have just one row for each PROVNUM. For the output here, I just show summary statistics on the AGESUM_PROVIDER file, using PROC MEANS (though I certainly could have used PROC SQL again.) See **Output 10**.

Exercise 2.6 - Facility summary of residents age 95+

The MEANS Procedure

Variable	N	Mean	Min	Max
NumRes	15527	22.64	1.000	211.0
Num_95plus	15527	1.766	0	28.00
Prop_95plus	15527	0.082	0	1.000

Output 10 Summary statistics on the data set that is produced by Exercise 2.6, facility-level summaries of resident age

Finally, before moving on to a new exercise, we add one more refinement to this one. Let's say we want to limit the analysis in the prior example just to facilities in the Northeast census region (CENSRegion = 1). This is very easily done by simply adding a WHERE clause, which must come immediately after the FROM expression.

```
TITLE3 'Exercise 2.6 - Facility summary of residents age 95+';
TITLE4 'Northeast region only';
PROC SQL ;
CREATE TABLE AGESUM_PROVIDER_NE AS
SELECT
    Provnum
   ,CensRegion
   ,N(AGE) AS NumRes
   ,SUM(Age_GE95) AS Num_95plus
   ,MEAN(Age_GE95) AS Prop_95plus
FROM in.ressamp2014
WHERE CensRegion = 1
GROUP  BY CensRegion, PROVNUM ;
QUIT;
```

We get the expected result here (one row per provider in the northeast census region) because, although we added CENSREGION to the SELECT clause, we also added it to the GROUP BY, and, of course, PROVNUM is nested within CensRegion. If we had added CensRegion to the SELECT and not to the GROUP BY, a "RE-merge" would have resulted, and the so-called summary data set would have a row for every RESIDENT in the northeast region. Again, We will come back to this.

The result (see **Output 11**) looks quite similar to Exercise 2.6, except, of course, the N (the number of facilities) is much smaller because it is just those in the Northeast.

Exercise 2.7 - Facility summary of residents age 95+

Northeast region only

The MEANS Procedure

Variable	N	Mean	Min	Max
NumRes	2626	31.08	1.000	211.0
Num_95plus	2626	2.914	0	28.00
Prop_95plus	2626	0.100	0	0.625

Output 11. Summary statistics on the data set that is produced by Exercise 2.7, facility-level summaries of resident age for nursing homes in the Northeast census region.

EXERCISE 3: COMBINING DETAIL AND SUMMARY DATA

It is no uncommon that one needs to combine some type of aggregate or summary information with the detail or individual-level data. You might want to compare each record in a sample to the overall mean, or the mean for specified groups, and that is the goal of this next set of exercises. PROC SQL can handle this type of task quite simply, while if you want to use PROC SUMMARY or MEANS to generate the summary statistic, you will then have to merge the summary statistics back with the individual data.

One of the variables on our nursing home resident data set is PHQ9, which is a depression score, and higher scores indicate greater depressive symptoms. Let's say we want to identify residents in each state that are above the mean for their state. In most previous examples with a GROUP BY clause, we have been careful to SELECT only the GROUP BY variable(s) and summary statistics, in order to avoid a "REMERGE" which results in a row being output not just for each group but for ALL the rows in the input. Here we want to take advantage of the "REMERGE" to combine the summary and detail information. The following code will create an indicator on each row of the resident sample of whether or not the PHQ9 value for that resident is higher than the state mean PHQ9.

```
TITLE3 'Exercise 3.1 - Add resident indicator of a depression score above
  the state mean';
PROC SQL ;
CREATE TABLE phqsum1 AS
SELECT *
    ,MEAN(phq9) AS StAvg_phq9 FORMAT=6.1
    ,(phq9 > CALCULATED StAvg_phq9) AS phq9_high
       LENGTH=3 LABEL='PHQ9 above state mean'
FROM in.ressamp2014
GROUP BY state
ORDER BY state, provnum ;
QUIT;
```

So, in addition to all the columns on the original RESSAMP file (the "*' syntax is a wild card referring to all columns of whatever entity is in the FROM clause), the new PHQSUM1 data set will have the state-specific mean for the PHQ9 (called StAvg_phq9) and a binary variable called PHQ9_high that has a value of 1 if the individual's PHQ9 score is higher than the state average and 0 otherwise. **Output 12** shows partial output of a PROC MEANS with state as a CLASS variable for the PHQ9 score and the new 0,1 indicator.

State abbreviation	N Obs	Variable	N	Mean	Std Dev	Min	Max
AK	157	PHQ9	152	2.862	4.074	0	19.00
		StAvg_phq9	157	2.862	0	2.862	2.862
		phq9_high	157	0.325	0.470	0	1.000
AL	5809	PHQ9	5556	1.514	2.657	0	21.00
		StAvg_phq9	5809	1.514	0	1.514	1.514
		phq9_high	5809	0.300	0.458	0	1.000
AR	4525	PHQ9	4297	1.986	3.134	0	26.00
		StAvg_phq9	4525	1.986	0	1.986	1.986
		phq9_high	4525	0.367	0.482	0	1.000
AZ	3123	PHQ9	2906	1.866	2.998	0	21.00
		StAvg_phq9	3123	1.866	0	1.866	1.866
		phq9_high	3123	0.341	0.474	0	1.000

Output 12. Partial output showing state average PHQ9 depression scores, state averages and proportion of residents above the average

Let's make a small refinement in this code. As noted above, the new column from Example 3.1, PHQ9_high will be 1 if the resident's PHQ9 score is higher than the state average and 0 otherwise. This "otherwise" includes the situation where the resident has a missing value for PHQ9. If we would like these individuals to be missing on the high score indicator, we can use the CASE expression, as shown in Exercise 3.2.

```
TITLE3 'Exercise 3.2 Refine resident indicator to account for missing
   data';
PROC SQL ;
CREATE TABLE phqsum2 AS
SELECT *
     ,MEAN(phq9) AS StAvg_phq9 FORMAT=6.1
     ,(phq9 > CALCULATED StAvg_phq9) AS phq9_high
        LENGTH=3 LABEL='PHQ9 above state mean'
     ,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 in.ressamp2014
GROUP BY state ;
QUIT;
```

I've find the CASE expression syntax to be a little confusing in that you don't specify the name of the column you are calculating (in this case PHQ9_high2) until the end of the expression, and the WHEN expressions provide the conditions and the values of the new column to be assigned for each of those conditions. The ELSE expression is the sort of "catch-all", explicitly stating what value to assign if none of the preceding WHEN conditions are met. The ELSE is not strictly required, but it is a good "defensive programming" technique. Aside from this one additional column, the resulting data set (PHQSUM2) is identical to the result of Exercise 3.1. Partial PROC MEANS output is shown in **Output 13**. Note that for each state, the numerators (the count of those with a 1 for the indicator – the SUM) are identical for the two indicators, but the denominators (the count of those with a non-missing value for the indicator) are different, with the PHq9_high2 indicator having a smaller value in each case. The N for this indicator matches that of the PHQ score (refer back to **Output 12**). Due to the smaller denominators, the proportions (Mean) are higher.

State abbreviation	N Obs	Variable	N	Sum	Mean	Min	Max
AK	157	phq9_high	157	51.00	0.325	0	1.000
		phq9_high2	152	51.00	0.336	0	1.000
AL	5809	phq9_high	5809	1742	0.300	0	1.000
		phq9_high2	5556	1742	0.314	0	1.000
AR	4525	phq9_high	4525	1660	0.367	0	1.000
		phq9_high2	4297	1660	0.386	0	1.000
AZ	3123	phq9_high	3123	1065	0.341	0	1.000
		phq9_high2	2906	1065	0.366	0	1.000
CA	26631	phq9_high	26631	5815	0.218	0	1.000
		phq9_high2	25273	5815	0.230	0	1.000

Output 13. Partial output showing the number and proportion of residents above the state average depression score, with two indicators, one that ignores missing data and the other that accounts for it.

A next step in this analysis of depression scores might be to select the residents with high depression scores. Of course, we could do this with an additional PROC SQL statement, selecting from the "re-merged" PHQSUM data set. However, we can also do it in a single step. Specifically, for Exercise 3.3, we want to create a new data set that contains only those residents that have a PHQ9 depression score at least 2 points above the gender-specific state mean. Here is the code:

```
TITLE3 'Exercise 3.3 - Select residents with PHQ9 scores 2 points above
   the state mean by sex';
PROC SQL ;
CREATE TABLE PHQ_high AS
SELECT *
    ,MEAN(phq9) AS StAvg_phq9_bySex FORMAT=6.1
    ,(phq9 > (CALCULATED StAvg_phq9_bySex + 2)) AS phq9_high1
FROM in.ressamp2014
GROUP BY state, sex
HAVING phq9_high1 = 1 ;
QUIT;
```

Let's pull this apart. The MEAN PHQ9 score that is calculated (StAvg_phq9_bySex) is calculated for each group defined by what is in the GROUP BY clause – here state, and sex. So, each state will have one average value for men and one for women. The indicator phq9_high1 for each resident will have a value of 1 if that resident's PHQ9 is 2 or more points above the gender-specific state average. The CALCULATED keyword is required because the State average value column is not on the input RESSAMP data set. Finally, the HAVING expression specifies which rows are written to the new PHQ_high dataset, those that meet the criteria of having the indicator with a value of 1. Note that you cannot use a WHERE clause here for two reasons. First, the PHQ9_high1 indicator doesn't exist on the input data set. However, even a WHERE clause with the CALCULATED keyword would give an error. The error you would get is "Summary functions are restricted to the SELECT and HAVING expressions only". This is a bit cryptic in a way, but you can see that the indicator is a result of a calculation on a summary function and so is considered a summary function. The code shown in Exercise 3.3 will work – though you will get a note in the log that a REMERGE occurred. **Output 14** shows just the first few of the selected rows.

Exercise 3.3 - Select residents with PHQ9 scores 2 points above the state mean by sex

Obs	STATE	sex	StAvg_ phq9_ bySex	PHQ9	phq9_ high1
1	AK	1	2.1	8	1
2	AK	1	2.1	10	1
3	AK	1	2.1	14	1
4	AK	1	2.1	7	1
5	AK	1	2.1	15	1
6	AK	1	2.1	7	1
7	AK	1	2.1	6	1
8	AK	1	2.1	6	1
9	AK	2	3.4	11	1
10	AK	2	3.4	7	1

Output 14. Partial output showing a few of the residents selected in Exercise 3.3, those with a PHQ9 score at least 2 points above the gender-specific state average.

One more little twist here. For Exercise 3.4, Let's say we want to select residents with PHQ9 scores 2 or more points BELOW the state average. If we just changed the '>' to '<' and the '+ 2" to "- 2" from Exercise 3.3, what would happen? You may have learned the hard way at some point that a missing value is a very, very large negative number – less than any actual value – so if we just made those minor tweaks to the

code, all those with a missing PHQ9 score would be selected. One way around this would be the CASE expression, but there is an even simpler way – the WHERE clause to select from the RESSAMP file first only those rows with a non-missing PHQ9 score. This doesn't affect the state averages since the missing values would not contribute. Otherwise, the code is very similar to Exercise 3.3 The code is shown below:

```
TITLE3 'Exercise 3.4 - Select residents with PHQ9 scores 2 points below
   the state mean by sex';
PROC SQL ;
CREATE TABLE PHQ_low AS
SELECT *
     ,MEAN(phq9) AS StAvg_phq9_bySex FORMAT=6.1
     ,(phq9 < (CALCULATED StAvg_phq9_bySex - 2)) AS phq9_low
FROM in.ressamp2014
     WHERE phq9 NE .
GROUP BY state, sex
HAVING phq9_low = 1 ;
QUIT;
////////////////////////////////////////////////////////////////////////////////
```

Note that SQL is very picky about the ORDER of the CLAUSES – and WHERE must come right after FROM, which makes some sense sit it specifying which rows of the FROM tables(s) the rest of the query will act on – Logically, it is good to think of this as the first thing that happens in the processing of the query. **Output 15** shows a few rows from the result.

STATE	sex	StAvg_ phq9_ bySex	PHQ9	phq9_low
AK	1	2.1	0	1
AK	1	2.1	0	1
AK	1	2.1	0	1
AK	1	2.1	0	1
AK	1	2.1	0	1
AK	2	3.4	0	1
AK	2	3.4	1	1
AK	2	3.4	0	1
AK	2	3.4	0	1

Output 15. Partial output showing a few of the residents selected in Exercise 3.3, those with a PHQ9 score at least 2 points below the gender-specific state average.

EXERCISE 4: NESTING SUMMARY FUNCTIONS

At first glance, the task for the next exercise seems very similar to the previous one. Let's say that we want to select the largest nursing home in each state, defined as the one with the most residents (that is, the largest number of rows in the RESSAMP data set.) Here's code that looks like it might work.

```
TITLE3 'Exercise 4.1 First attempt to select nursing home with largest
   number of residents in each state';
PROC SQL ;
CREATE TABLE largest as
SELECT provnum, state, COUNT(*) as numres
FROM in.ressamp2014
GROUP BY state, provnum
  HAVING NUMRES = MAX(NUMRES) ;
QUIT;
////////////////////////////////////////////////////////////////////////////////
```

In the SELECT clause we are choosing the facility identifier, the state and the count of residents (rows) for each state. The Group BY specifies that the summary function will group the rows by state and by provider within state. And the HAVING expression specifies to output to the dataset LARGEST the rows within each state that have the number of residents (for that provider) equal to the larges value of that summary function. However, the code doesn't work – we get the following ERROR message: "Summary functions nexted in this way are not supported". What is meant by nested summary functions? Well, in the first part of the HAVING expression, the NUMRES refers to the count of residents in each nursing home – which was obtained by a summary function (the COUNT *) grouped on stated and provider number. In the second part of the HAVING expression, we are asking SQL to compute a summary function (getting the MAX value) on that column NUMRES, which is already a SUMMARY function result – hence, they are nested.

So, how do we get around this to select the largest provider? One way to do it is in two steps, first calculating the number of residents in each nursing home and then, in a second query selecting the provider with the largest count. This is shown in Exercise 4.2, and it works perfectly well.

```
TITLE3 'Exercise 4.2 - Select largest nursing home in each state in two
   steps ';
TITLE4 'Step 1 - Count residents in each nursing home';

PROC SQL ;
CREATE TABLE COUNTRES AS
  SELECT state
         , provnum
       ,COUNT(*) AS numres
FROM in.ressamp2014
GROUP BY state, provnum ;
QUIT;

TITLE4 'Step 2 - Select nursing home(s) with largest number of residents';
PROC SQL ;
CREATE TABLE largest_V1 AS
 SELECT *
 FROM countres
 GROUP BY state
 HAVING numres = MAX(NUMRES);
 QUIT;
```

After Step 1, the data set COUNTRES will have one observation for each nursing home id (which is nested within state) that contains the variable numres, which is the row count from the RESSAM data set. Then, in step 2, we read from the COUNTRES data set, grouping BY state and selecting the rows where NUMRES (a summary function from Step 1) is equal to a new summay function – i.e. the maximum value of NUMRES for each GROUP BY group – here, state. Note that both queries could be in the same PROC SQL step – two separate calls to the PROC are not required – one can contain multiple queries. **Output 16** shows a few rows from the resulting data set. Note that the file might have more than one row for each state – if two are more nursing homes are "tied" for the largest number of residents

```
Exercise 4.2. Select the nursing home with the most residents in each state

                          STATE    PROVNUM    numres

                           AK      025025        27
                           AL      015031        62
                           AR      04A293        55
                           AZ      035145        62
                           CA      555020       192
                           CO      065379        53
```

Output 16. Partial output showing a few of the nursing homes selected in Exercise 4.2 – those with the largest number of residents in each state.

So, that achieved our task – but it is possible to do it in a single query. We are forbidden to next summary functions as we learned from our first attempt at this task. However, we can "nest" queries. What we do is basically put the query from Step 1 above into the FROM clause of the query from Step 2. Here is the precise code:

```
TITLE3 'Exercise 4.3 Select largest nursing home in each state -one step ';
 PROC SQL ;
 CREATE TABLE LARGEST_V2 AS
 SELECT *
 FROM
  (SELECT provnum, state
       ,COUNT(*) AS numres
   FROM
    in.ressamp2014
   GROUP BY state, provnum)
GROUP BY state
HAVING numres = MAX(numres) ;
QUIT;
```

When nesting one query inside another like this in the FROM clause, it is required to put this neseted query (called an inline view) in parentheses. So, before SQL can execute the outer query, it must do the summarization requested in the inline view and then this "virtual" table becomes the source SQL pulls from in doing the 'outer' summarization and generating the LARGEST_v2 data set. The result is identical to what we achieved in two queries in Exercise 4.2 – refer to **Output 16**.

The final part of this exercise is another example of using an inline view. Here we want to select the nursing home in each state with the largest number of minority residents. There is already a variable MINORITY that is a 0,1 indicator for each resident of being a racial ethic minority. The code for Exercise 4.4 demonstrates this second example of an inline view.

```
TITLE3 'Exercise 4.4 - Select nursing home(s) with greatest proportion
  minority in each state';
PROC SQL ;
 CREATE TABLE minority1 AS
  SELECT *
  FROM
  (SELECT STATE, PROVNUM
      ,MEAN(minority) AS prop_minority
    ,COUNT(*) AS numres
   FROM in.ressamp2014
    GROUP BY state, provnum)
```

```
GROUP BY STATE
HAVING prop_minority = MAX(prop_minority)
ORDER BY state, prop_minority DESCENDING, numres DESCENDING;
QUIT ;
```

Within the inline view we use a summary function to get the proportion of residents in each nursing home who are minority by taking the mean of the 0/1 indicator. This virtual table has a "row" for each nursing home. Then the desired table (MINORITY1) is selected by selecting the rows from that virtual table that meet the criterial of having that proportion equal to the largest value of it, defined at the level of the Group by clause in the outer query – that is, state. Just a few rows of the result are shown in **Output 17**.

Exercise 4.4 - Select nursing home(s) with greatest proportion minority in each state

STATE	PROVNUM	prop_ minority	numres
AK	025037	1.00	4
AK	025035	1.00	3
AK	025026	1.00	3
AL	015043	0.95	20
AR	045334	1.00	23
AR	045364	1.00	22
AR	045398	1.00	14
AR	045194	1.00	13
AR	045365	1.00	7
AZ	035216	1.00	23

Output 17. Partial output showing a few of the nursing homes selected in Exercise 4.4 – those with the largest proportion of minority residents in each state.

EXERCISE 5: GENERATING SUMMARY STATISTICS ON DERIVED COLUMNS

In addition to the facility to combine summary and detail information in a single step, another great feature of PROC SQL summary functions is the capability to compute summary statistics on newly created or derived variables – ones that do not exist on the input data set but can be derived from existing variables. PROC SUMMARY/MEANS-based techniques would require pre-processing to achieve this.

Let's say that we want to summarize the resident-level file to the facility-level with a count of the number of men and the number of women taking antipsychotics. Our goal is to produce a file with one row per nursing home (PROVNUM). A first attempt (shown below) might just add SEX to the GROUP BY clause as shown in the second PROC SQL step below

```
* generate a file with one record per provider with the count and
  proportion of residents using Antipsychotics ;
TITLE3 'Exercise 5.1 - Review - do for both genders combined ';
PROC SQL ;
CREATE TABLE antipsych1 AS
SELECT provnum
       ,sum(antipsych) AS Num_antipsych
     ,mean(antipsych) AS prop_antipsych
FROM in.ressamp2014
GROUP BY provnum ;
QUIT;

* To get a gender-specific count, first try just adding SEX to the SELECT
```

```
   and GROUP BY clauses;
TITLE3 'Exercise 5.2 - Try just adding sex to the GROUP BY ';
PROC SQL ;
CREATE TABLE antipsych2 AS
SELECT provnum, sex
       ,sum(antipsych) AS Num_antipsych
     ,mean(antipsych) AS prop_antipsych
FROM in.ressamp2014
GROUP BY provnum, sex ;
QUIT;
```

By now you probably realize that GROUP BY generates a row for each unique combination of the GROUP BY variables. So the code above runs without error but it will generate a row for men and another for for women for each nursing home (except if a nursing home has only men or only women). Indeed, the log message we get indicates that the data set ANTIPSYCH2 has 30576 rows, which is close to twice the number of nursing homes in the sample (N=15,527). So, what can we do to get the desired counts but generate just one row per nursing home? If we think about the problem a little differently, we realize that what we want to count is the combination of two variables – SEX and the Boolean ANTIPSYCH, another approach becomes clear – we can count the combinations that meet the criteria of interest. See the code below:

```
TITLE3 'Exercise 5.3 - Count two new indicators that combine sex and
  Antipsych ';
PROC SQL ;
CREATE TABLE antipsych3 AS
SELECT provnum
      ,SUM(sex = 1 AND antipsych = 1) AS men_Antipsych_n
      ,MEAN(sex = 1 AND antipsych = 1) AS men_Antipsych_pct
      ,SUM(sex = 2 AND antipsych = 1) AS women_Antipsych_n
      ,MEAN(sex = 2 AND antipsych = 1) AS women_Antipsych_pct
FROM in.ressamp2014
GROUP BY provnum ;
QUIT;
```

Note that in order to avoid the REMERGE which would produce a data set with a row for every resident, we must NOT add the "virtual" combination variables (e.g. sex = 1 and antipsych = 1) to the SELECT clause. Instead, we select only the GROUP BY variable (PROVNUM) and columns that are summary functions Output 18 shows PROC MEANS results for the new summary columns. Note that the N for each is 15,527 – the number of nursing homes.

Exercise 5.3 - Count two new indicators that combine sex and Antipsych
The MEANS Procedure

Variable	N	Mean	Std Dev	Minimum	Maximum
men_Antipsych_n	15527	1.9010111	2.9931526	0	57.0000
men_Antipsych_pct	15527	0.0810989	0.1050973	0	1.0000
women_Antipsych_n	15527	3.0104978	3.0129961	0	47.0000
women_Antipsych_pct	15527	0.1312385	0.1066528	0	1.0000

Output 18. Results from Exercise 5.3, generating nursing home summary statistics for antipsychotic use for men and women.

Another common example for which you might want to be able to generate summary statistics on a derived measure would be if we wanted to get the average age of all residents on a certain date (or if an age variable was not already on the input data set. Let's say we want to calculate the average age of residents

by state on December 31,2014 – rather than using the age variable already on the data set. Here is the code:

```
TITLE3 'Exercise 5.4 - Summary statistics on Resident Age on 12/31/2014 by
   State ';
PROC SQL ;
CREATE TABLE AgeDec2014 AS
SELECT  State
       , MEAN( YRDIF(DOB,MDY(12,31,2014),'AGE' )) AS mean_age
FROM in.ressamp2014
GROUP BY State ;
QUIT;
```

And a partial listing of the output is shown in **Output 19**.

Exercise 5.4 - Compute summary statistics on Resident Age on 12/31/2014 by State

STATE	mean_age
AK	76.1925
AL	77.9765
AR	79.5447
AZ	75.1078
CA	76.7785
CO	79.3474
CT	81.3726
DC	76.1527
DE	79.8557
FL	79.5803

Output 19. Partial listing of results from Exercise 5.4, generating state summary statistics for resident age on a specified date

EXERCISE 6: GENERATING MACRO VARIABLES BASED ON SUMMARY STATISTICS

You can also use PROC SQL to pass summary information from a data set to macro variables using the INTO: syntax. The first part of this exercise generates counts of the total number of rows (residents) in the resident sample data and the total number of nursing homes. Note that the DISTINCT syntax means that a count of unique provider numbers is generated. Also, the NOPRINT option on the PROC SQL statement prevents sending any output to open output destinations.

```
TITLE3 'Generating macro variables based on summary statistics';
TITLE4 'Exercise 6.1: Total resident count and facility count';
PROC SQL  NOPRINT ;
SELECT  PUT(COUNT(*),COMMA8.)
       ,PUT(COUNT(DISTINCT PROVNUM),COMMA6.)
INTO :totres
    ,:totfac
FROM in.ressamp2014 ;
QUIT;

TITLE5 "The sample contains &totres. residents in &totfac. facilities";
PROC MEANS DATA = in.ressamp2014 ;
VAR phq9 ;
RUN;
```

This code generates two macro variables &totres and &totfac, which could be used in later processing, or, most simply in a TITLE statement as shown. The results are shown in **Output 20**.

```
          Exercise 6.1: Total resident count and facility count
         The sample contains  351,570 residents in 15,527 facilities

                              The MEANS Procedure

                  Analysis Variable : PHQ9 Resident Mood Score (PHQ-9)

      N            Mean         Std Dev         Minimum         Maximum
ƒ──────────────────────────────────────────────────────────────────────
335813       2.5173832       3.6383156               0      30.0000000
ƒ──────────────────────────────────────────────────────────────────────
```

Output 20. Demonstrating the creation of macro variables based on summary statistics

Imagine that there is some process you need to run for every provider, and you are going to write a beautiful macro to do it. Wouldn't it be nice to be able to pass the provider identifiers to that macro "automatically". One way to do this would be to generate the macro variables using CALL SYMPUT in a data step – another way to do it is with PROC SQL. In this exercise, we put the provider numbers of all the facilities in Alaska into macro variables. Of course, this could easily be translated to other applications. Again we use DISTINCT so that only one macro variable is generated per unique PROVNUM. Also, note that it is not a problem to generate "extra" macro variables – they simply don't get any values if there are more than there are providers. Output 21 shows what is written to the log with the %PUT statement.

```
PROC SQL NOPRINT ;
SELECT distinct provnum
INTO :facid1 THROUGH :facid25
FROM in.ressamp2014
   WHERE state = 'AK'    ;
QUIT;

%PUT &facid1 &facid2 &facid20 ;
```

```
WARNING: Apparent symbolic reference FACID20 not resolved.
                  025010 025015  &facid20
```

Output 21. SAS Log: Demonstrating the creation of macro variables based on provider identifiers

CONCLUSION

PROC SQL can do a wide variety of data summarization/aggregation tasks – either for the purpose of generating simple descriptive statistics or as a necessary step in data management tasks. Thoughtful, deliberate use of the REMERGE and the ability to produce summary statistics on derived variables allow one to accomplish in a single query what might require pre- and/or post-processing if you rely on PROC SUMMARY. And with inline views you have seen how you can nest summary functions. I hope that these step by step examples have made using SQL summarization a little more transparent. Good luck!

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*

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

Williams, Christianna. “PROC SQL for PROC SUMMARY Stalwarts.” Proceedings of SAS Global Forum 2015. Available at http://support.sas.com/resources/papers/proceedings15/3154-2015.pdf

CONTACT INFORMATION

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

Christianna Williams
Christianna.S.Williams@gmail.com