

A Simplified Reporting Strategy: "summary" data sets for reporting on data from multiple source data sets

**Steve Ross, ClinTrials Research, Inc.
Research Triangle Park, NC**

In conducting analysis of clinical trials data, or in any programming environment, it is rare that all of your data will come from one source data set. In the interest of simplicity and efficiency, a single "summary" file created early in the programming process and containing stratification variables applicable to many tables is strategically beneficial to the programming, reporting, and QC processes. Placing such key values in a central location gives the programmer significant advantages. From a strategic standpoint, programming from a single data source is simpler than managing variables from three or four data sets, especially those of differing structure. Data-checking and QC are also made easier given the single data set. In addition, the summary file can easily be merged onto data sets, giving the user all relevant stratification variables in one step. With some planning and forethought as to the requirements of the analysis, the summary data set can make life easier for any programmer. Written for the novice programmer, this real-world example steps through the design and creation of a summary data set, and the use of sound programming strategies to achieve the most usable information.

Introduction

The analysis of data can be a cumbersome and uncertain process in the absence of clear ground rules. Especially so in the clinical trials business, where data arrives on "Case Report Forms" and are typically entered into 10 to 40 different data sets. The key, from a programming perspective, is to separate the immediately "important" stratification data from the larger database and organize it to make it most usable. Simply put, identify the key variables you'll need for your output, and put them in a separate summary data set that is structured according to the needs of the analysis plan. For most purposes a data set with one record per unique id is most useful. In doing this you'll have a valuable tool that is ready-made for much of the output you will produce. The example used in this paper covers a clinical trial, though the concept of a summary file may be of benefit to anyone who programs using a large volume of data that are dispersed among many data sets of differing structure and content.

Objectives/Ideas

One way we can begin the design and programming of our summary file is with answers to the following questions:

1) What is the analysis objective?

2) Will the data available to us give us the information we need?

3) Do the data need transformation to fit into our summary file?

Armed with the knowledge from the answers to these questions, we'll create a finished data set ready to use in final output reporting.

1) What is the analysis objective?

As noted earlier, for the purpose of this paper our programming assignment will be to report on a clinical trial, a hypothetical study of a blood pressure lowering drug.

First off, one question we'd like to answer is if some subgroup of patients react differently to a drug than other populations. In the analysis of clinical trials, as in other fields, the "important" variables often include demographic stratification information (age, sex, race, etc). So we'll keep only those variables from the demographics data set (including the unique keys as well). In addition, variables that describe the efficacy and safety effects of the tested drug on the body are needed as well. From an efficacy standpoint, we'd like to know how a patient's blood pressure changed during the treatment period, and what dose of drug a patient has to take to achieve a certain level of blood pressure "lowering".

And from a safety standpoint, it is important to ascertain whether the effects of the drug are long-lasting, and if there are any side-effects, especially with increasing number of doses administered.

So, taking the above as a crude analysis plan, we can clearly see what information is "important", and what should be in our summary file. There will be subsetting based on the demographic variables sex, race, and age, so those should be included. To answer the efficacy question we'll need to have blood pressure readings from a vital signs data set; to address safety concerns we'll pin down the dates of treatment, i.e. start and stop dates from the dosing file, in addition to how much drug was given at each interval. Ultimately we'd like to get all of this information into one observation per unique id (in this case one per individual patient), so that our summary data set will be easily used, both as a standalone analysis data set and in conjunction with other data sets.

Data that would be collected typically include, in addition to the above-mentioned information, medical history data, adverse events, and concomitant medications. However, as they tend to contain an open-ended number of observations for each patient, it is probably unfeasible to place them in our summary file and maintain the one record per patient standard. Attempting this could result in the data set becoming impossibly "wide", having many more variables than necessary, many of which would contain blanks. So, in this case, they will not be included in the summary file. To jump ahead a bit, however, summarizing this type information will be made easier by being able to merge a completed summary file to these data sets in the analysis process.

2) Will the data available to us give us the information we need?

While this may seem like a somewhat pointless question, it's far better to ask it before programming begins than to find out after substantial work has been done. Perhaps the singular most important caveat to any programmer is "know thy data", and this is the best reason to do so.

The base SAS® system offers many standard tools by which to become intimately acquainted with your data, and, in general, using SAS procedures (PROCs) requires less programming than SAS data steps. A few of the most applicable procedures for our next task, *data exploration*, follow:

- PROC CONTENTS provides a summary of the structural configuration of your data, including the data set name, how many observations (rows) and variables (columns) your data set contains, and the names, formats, informat, and labels of each of the variables on the data set. PROC CONTENTS will also reveal whether or not your data is sorted, and, if so, list the sort order of the variables. A well constructed data set will have labels for all variables that give a clear picture of the data that reside in them. Keep this in mind when constructing your summary file later. The CONTENTS procedure is easily invoked for one data set or all data sets in a library, after issuing a LIBNAME statement:

```
LIBNAME sasdata "C:\mydata";

TITLE1 "PROC CONTENTS - 1 data set";
PROC CONTENTS data=sasdata.demog;
RUN;

TITLE1 "PROC CONTENTS - all data sets in a library";
PROC CONTENTS data=sasdata._all_;
RUN;
```

Once you've gotten this information, it's easy to dig deeper:

- PROC FREQ, at the most basic level, will provide you with counts and percentages of any variable you specify in the TABLES statement. This is particularly helpful in the case of data that contains many categorical variables, such as might be required in a summary data set like sex and race. PROC FREQ gives you a basic feel for how your data break down according to your stratification variables, the very variables you'll want in your summary files. The syntax is also simple. Using the LIBNAME issued previously, simply submit the following:

```
TITLE1 "PROC FREQ output for sex and race
variables in the Demographics data set";
PROC FREQ DATA=sasdata.demog;
TABLES sex race;
RUN;
```

The output returned shows the counts and percents of each gender and race in the data set. More detailed information results from simply changing the TABLES statement:

```
TITLE1 "PROC FREQ of sex variable, by race";
PROC FREQ data=sasdata.demog;
TABLES sex * race;
RUN;
```

Submitting these statements returns a table detailing the counts and percentages of each sex, now stratified by race.

•PROC MEANS produces simple descriptive statistics for each numeric variable of a data set (means, standard deviations, minimum, maximum) as well as the number of non-missing observations in each variable. Essentially the same as PROC SUMMARY, PROC MEANS, by default, produces printed output where SUMMARY does not. PROC MEANS is invoked simply through the following code:

```
TITLE1 "PROC MEANS Output";
PROC MEANS DATA=sasdata.demog;
CLASS SEX;
RUN;
```

Note the use of the CLASS statement instead of a BY statement, which allows for the grouping of data without first having to sort the data in the order of the class variables. Generally, this is preferable to using BY, as calling PROC SORT is a memory intensive and time consuming PROCEDURE.

•PROC UNIVARIATE does what its name implies, producing univariate distribution statistics on all *numeric* variables or for a variable of interest in the data set specified. This is particularly useful in the data checking process, if your goal is to locate outliers and missing information. PROC UNIVARIATE provides mean information, but also calculates median/mode information, quantiles, range information, and can identify the five

highest and lowest values of each variable. This makes PROC UNIVARIATE an excellent companion to PROC MEANS, as the UNIVARIATE PROCEDURE builds on the information provided by MEANS. All that's needed for this procedure is the following:

```
TITLE1 "A sample of PROC UNIVARIATE
output for one data set";
PROC UNIVARIATE DATA=sasdata.demog;
RUN;
```

or, for all data sets in a SAS Library:

```
TITLE1 "A sample of PROC UNIVARIATE
output for all data sets in a library";
PROC UNIVARIATE DATA=sasdata._all_;
RUN;
```

3) Do the data need transformation to fit into our summary file?

Any application or piece of code requiring significant processing of data is made easier if irrelevant information is removed. But before going down this path, an important observation should be made regarding data transformation. Remember that any changes you make will be imposed on the dataset permanently, so wherever you work, make certain that you are not overwriting the original data sets by using OUT= options for PROCedures and the WORK data area or some other fully qualified two level SAS name (SASDATA.DEMOG) for DATA steps:

```
TITLE1 "PROC SORT writing to the WORK
library";
PROC SORT DATA=sasdata.demog
          OUT=work.demog;
BY ID;
RUN;
```

and

```
TITLE1 "DATA step writing demog from
sasdata library to WORK library";
DATA work.demog;
  SET sasdata.demog;
RUN;
```

This ensures that you are working with copies of your data, rather than replacements. That being said, in data exploration the two simplest methods of

doing this are dropping/keeping variables, or subsetting out unneeded observations. These are accomplished through the use of IF, DROP, KEEP, and WHERE statements and DATA step options. In general, use DROP and KEEP when your goal is to eliminate variables (columns), and WHERE to eliminate observations (rows):

```
/* Using KEEP and WHERE */
DATA work.demog;
  SET sasdata.demog (KEEP= id sex race
                      WHERE=(race='O'));
RUN;
```

```
/* Using DROP and IF */
DATA work.demog;
  SET sasdata.demog(DROP=initls
                    shoesize);
  IF eyecolor='green';
RUN;
```

Two things about these short pieces of code are notable. First, using DROP and KEEP in the SET statement is more efficient than the same DATA step option applied to the DATA statement (or in open code), as the constraints are applied before the observation of data is read into the Program Data Vector (PDV). Only the needed variables are placed in the PDV, so I/O, memory, and processing time are minimized¹. The resulting data set is the same whether the WHERE clause is applied to the target or source data set, but this simple efficiency tip pays off the larger your data sets are. The same rule applies to the subsetting IF in the second example. A WHERE clause applied in the same manner as the first example would be more efficient. In that example all observations are kept until that statement is executed, even though only patients with eyecolor="green" are needed for this particular data step. A more efficient DATA step would be coded as follows:

```
DATA work.demog;
  SET sasdata.demog(DROP=initls
                    eyecolor
                    WHERE=(eyecolor="green"));
RUN;
```

Structural transformations of your data are also occasionally needed, depending on the requirements of your project. In essence, this generally means converting a data set

from many observations per unique id variable to a single observation, or vice versa. There are a few variations on that theme, but the most common can be handled with the following methods:

•PROC TRANSPOSE, at most basic level, converts rows to columns and columns to rows. For example, the following code does just that for our data set sasdata.demog (Note - the manipulation below is simply for illustration - the *original* structure of this data set is the form we'd like for our summary file);

```
/* Converting rows to columns */
PROC TRANSPOSE DATA=sasdata.demog
  OUT=work.demog;
RUN;
```

before:

id	sex	age	race	weight	study
11	m	56	C	170	A
12	f	61	O	150	A
13	f	49	H	135	B
14	m	68	A	154	A

after:

col1	col2	col3	col4	col5
id	11	12	13	14
sex	m	f	f	m
age	56	61	49	68
race	C	O	H	A
weight	170	150	135	154
study	A	A	B	A

PROC TRANSPOSE, however, gets rapidly more complicated, and more complex data set transpositions require careful attention to the results, as they can be unpredictable even to the most experienced of programmers. For now, as those more complex transactions are beyond the scope of this paper, we'll leave them alone.

Another common transformation requires taking one observation and breaking it into many, and is also easily accomplished. If, for example, a data set containing Blood Pressure readings for 2 courses (baseline and 2 post-treatment measurements) of a trial is structured as follows, use a simple DO loop to separate it into one observation per blood pressure reading (again, we'll use the original form of this data set to maintain our one record per patient standard and simply illustrate the principle):

before:

id	bp1	bp2	bp3
11	115/70	110/80	125/60
12	120/80	125/85	130/75

```
DATA work.vitals (KEEP=id bp);
  SET sasdata.vitals;

  ARRAY bps(3) bp1-bp3;
  DO i=1 to 3;
    bp=bps(i);
    reading=i;
  OUTPUT;
  END;
RUN;
```

after:

id	bp	reading
11	115/70	1
11	110/80	2
11	125/60	3
12	120/80	1
12	125/85	2
12	130/75	3

At this point, we've chosen the variables we'll need through careful reading of the analysis plan, we've used some basic exploration PROCedures to get to know our data, and we've done some simple transformations to weed through extraneous data further. Now that we've illustrated these basic techniques, just merge them together using common keys; in this case only id is necessary.

```
/* MERGE demog and vital signs dataset,
   and write out bad records */
DATA sasdata.all work.nodememo
work.novital;
  MERGE work.demog(IN=demo)
        work.vitals (IN=vital);
  BY id;

  IF NOT demo THEN OUTPUT nodememo;
  IF NOT vital THEN OUTPUT novital;
  OUTPUT all;
RUN;
```

Note the use of the IN= option, which is helpful, in our scenario, for determining when key data are missing. Since we've identified the variables we're working with as "key" to our analysis, we'll need to know if any are missing. The IF statements direct observations to separate datasets in the case of missing information. Simply print out work.nodememo, and work.novital to

check for holes in your data, or check the SAS logs for notes indicating nodememo, nodose, or novital have more than zero observations.

Finally we've got a data set that is essentially a chronological chart of the patient's progress through the study, all on one line. It can be used in a standalone fashion, or simply merged onto the datasets we mentioned earlier as being unfeasible for use in our summary dataset.

Now, we're almost finished, but a last tool can be used to gain some efficiency. INDEXing a dataset allows SAS to access individual data points much faster than a data set without an index. However, the decision of whether or not to index is dependent on a number of factors, most notably the size of the data set and how often the data set is modified. Good general rules of thumb to drive the decision of whether or not to index are the following: 1) how large is the data set? and 2) how often does it change? A data set that has many thousands of observations and is not updated often is a good candidate for indexing. The smaller a data set is, the less advantage is gained by indexing. For illustration purposes, and though our data set is small, a sample DATASETS PROCEDURE is coded below, creating a simple index based on our unique identifier id:

```
/* Create a simple INDEX on patient id */
PROC DATASETS LIBRARY=sasdata;
  MODIFY all;
  INDEX CREATE id;
RUN;
QUIT;
```

Conclusion

Over the course of this paper, we've outlined how to choose the variables that would fit into a summary file based on the requirements of the analysis plan, how to do some straightforward data checking with PROCedures provided by the base SAS System. The IN= DATA step option was introduced as another data checking mechanism while all of the data sets were match-merged. Similarly, we've built in some efficiency measures to speed processing time of our summary file (using

WHERE rather than IF, and INDEXing). In conclusion, the concept of a summary file is one which any SAS user may find useful, especially if programming from multiple source data sets is commonplace; the ideas presented in this paper are stepping stones to creating them.

Raw Data

SASDATA.DEMOG					
id	sex	age	race	weight	study
11	m	56	C	170	A
12	f	61	O	150	A
13	f	49	H	135	B
14	m	68	A	154	A

SASDATA.VITALS			
id	bp1	bp2	bp3
11	115/70	110/80	125/60
12	120/80	125/85	130/75
13	115/75	130/80	130/80
14	140/80	145/85	140/85

Final Summary Data

id	sex	age	race	weight	study	bp1	bp2	bp3
11	m	56	C	170	A	115/70	110/80	125/60
12	f	61	O	150	A	120/80	125/85	130/75
13	f	49	H	135	B	115/75	130/80	130/80
14	m	68	A	154	A	140/80	145/85	140/85

Acknowledgements

SAS® is a registered trademark of SAS Institute, Inc. in the USA and other countries. ® indicates USA registration.

References

¹ SAS Institute Inc., SAS® Programming Tips: a Guide to Efficient SAS® Processing, Cary, NC: SAS Institute, Inc., 1990.

The following people contributed to the preparation of this paper:

Phyllis O'Hara
Syamala Kasichainula
Meredith Nahm

Questions, comments, or requests for reprints of this paper should be directed to:

Steve Ross
ClinTrials Research Inc.
P.O. Box 13991
Research Triangle Park, NC 27709
(919) 462-2814
sross@clintrialsresearch.com