Learning About Your Data: Tips and Techniques for Looking at Large Files

Sandra D. Schlotzhauer, Schlotzhauer Consulting
Bob Anschuetz, Quintiles, Inc.

ABSTRACT

As a SAS® consultant, you often receive unfamiliar data. A first step in analyzing or reporting on the data is to familiarize yourself with the data. For small files, or only a few files, simply printing the data may suffice. As the number of files and size of each file increases, this becomes increasingly less effective. We cover several techniques to investigate large files including: checking the structure of the data sets, finding duplicate variables, determining if the data values are reasonable, finding *holes* (unexpectedly large numbers of missing or identical values), using formats to handle long character values, using indicator variables instead of numeric variables, and more. We also cover general strategies, such as documenting the data sets as received, documenting changes, using subsets of the data for testing, and keeping the original data sets intact. Many of these tools are useful even for cases where you cannot change the original data, since the results give you a better understanding of the data. Examples are primarily from the pharmaceutical industry, but the techniques can be useful for other types of data as well.

INTRODUCTION

We first discuss general strategies and recommendations for working with data sets. This includes using PROC CONTENTS to familiarize yourself with the data, using labels for data sets and variables, and documenting revisions to the data.

Next, we cover issues in checking your data set structure. This includes finding if your data is sorted, confirming the number of records, and strategies for when to split a large data set into multiple smaller data sets.

Next, we cover checking the variables in the data sets. This includes checking for the same information stored in different variables and consolidating when possible, using character variables as indicators instead of using numeric variables, and appropriate use of formats.

Next, we cover checking values for data. This includes determining if the individual data values are reasonable for a variable, determining if mathematical combinations of variables are reasonable, and finding *holes*, or unexpectedly large numbers of missing or identical values for a variable. The paper closes with a brief summary.

Many of the tips and techniques are accompanied by code samples. These have been run on some sample data using Release 6.11. For straightforward examples, such as sorting, the code samples are generic enough that they can be applied to other data. For more detailed examples, such as checking the number of records per-X-per-Y (for example, the number of records per patient-per visit), the code samples are specific to the data. You probably need to revise them to work with your data.

GENERAL STRATEGIES

This section covers general recommendations for working with data sets, regardless of size. However, given that many issues are magnified with larger data sets, we consider these just as important as special large-size techniques.

General Strategies

- Do not overwrite original data sets.
- Use PROC CONTENTS to document original data.
- Label data sets and variables.
- Document data set revisions.
- Use subsets of data during development.
- Consider WRITE or ALTER passwords.

Do Not Overwrite Originals

Always keep the original data sets as you received them. If you find a programming error and need to start all over again, you'll need the original data sets as a starting point. If you need to document how the data sets changed during the course of your work, you'll need the original data sets as a starting point. We recommend keeping the originals on your system, and if you use a PC, keeping a second copy of the originals on diskette. This way, even if your PC hard drive crashes, you still have a copy of the originals. If you work on a network or mainframe system, check to be sure that the information is included in regular system backups. As an example, on a UNIX system, a TMP directory may not be included in automatic backups, so this is not a good location to store your original data.

Tip: To ensure that you do not inadvertently overwrite the original data sets, always use two-level data set names. Further, use descriptive libref assignments in your LIBNAME statement. For example, consider the two sets of code below:

```
proc sort data=a.demog out=b.demog;
   by patid aecode aestdt;
run;
proc sort data=origs146.demog out=cpys146.demog;
   by patid aecode aestdt;
run;
```

The librefs in the second set of code are much more likely to convey to you (or another programmer) that the original data set is being sorted and output to a different library. Using origs146 instead of a as a libref helps convey this information quickly.

Run PROC CONTENTS on Original Data

When you first begin to look at the original data sets, run PROC CONTENTS and save the output. We recommend printing the output and also saving it to a file. The output from PROC CONTENTS serves as documentation of the data sets as you received them. This can be very useful in clarifying misunderstandings about the data sets, checking if the data sets are as expected, and checking your results.

Further, if you receive the data sets in SAS transport files, ask the sender to ship the PROC CONTENTS output for the original data sets. Why? When you CIMPORT the transport files, the sort flags are not retained. If the sender's data sets are sorted and have the sort flags set, you can discover this in their PROC CONTENTS output. One of your first actions is then to sort the data so that you start with data sets in the same sort order as your customer intended.

Use Labeling of Data Sets and Variables

Suppose you do have the ability to change the data sets (always changing a copy of the originals!). If the original data sets or variables do not use labels, then you can add them. By adding labels to data sets and variables, you can convey much more information than in an 8-character name. Next year, when you need to return to this data, you'll be glad for the labels!

For example, suppose you need to sort the original data. In addition, you want to add a label to the data set. The following example code performs both these actions:

```
proc sort data=origs146.demog
     out=cpys146.demog(label=
         'Patient Demographics');
   by patid aecode aestdt;
run;
```

Tip: One idiosyncracy of data set labels is that they are not retained when you create a new data set with a DATA statement. So, for example, the following code loses the data set label for DEMOG:

```
data work.demog;
   set cpys146.demog;
   where age<65;
run;
```

The following code keeps the data set label, and also uses the WHERE= data set option instead of using a separate WHERE statement:

```
data work.demog(label='Patient Demographics');
   set cpys146.demog(where=(age<65));
run;
```

Of course, you can use a different label for the WORK data set to indicate that the data set includes only subjects under 65 years old.

Although labeling of data sets is useful, you typically only deal with data set labels in procedures such as PROC CONTENTS. Labeling of variables is far more useful since the labels are used by many procedures. This can happen either automatically (in PROC FREQ for example) or with an option (in PROC PRINT for example).

Although you can label variables temporarily for the output from just one procedure, it is much more useful to label the variables in a DATA step. This keeps the label associated with the variable for all procedures. In labeling variables, you can use either the LABEL or ATTRIB statement. The LABEL statement simply lets you assign labels to the variable. The ATTRIB statement also lets you assign formats, informats and lengths for the variable.

Tip: In general, if you know that the only attribute of a variable that you are likely to change is the label, just use the LABEL statement. If you think you might need to later change something else about the variable, then use an ATTRIB statement. The decision isn't permanent. Since you document changes with PROC CONTENTS, you can use first a LABEL statement and change later to an ATTRIB statement if needed. The following code gives examples of using both a LABEL statement and an ATTRIB statement:

```
data work.demog;
   set cpys146.demog;
   label Age='Age in Years';
   attrib agerg length=$8
        label='Age Categories';
   if age=. then agerg='Missing';
   else if age<65 then agerg='Under 65';
   else if age>=65 then agerg='Senior';
run;
```

The LABEL statement simply assigns a label to the existing AGE variable. The ATTRIB statement assigns a length and label for the new AGERG variable, which is created in the DATA step. Without the length assignment in the ATTRIB statement, SAS sets the length of the AGERG variable to be that of the first value it sees. This would be fine for the values “Missing” and “Senior” but the value “Under 65" would be truncated to read “Under 6"—a very different interpretation!

Document Data Set Revisions

When your programs modify data sets, document your changes. One way to do this is to use comments in your programs that identify the changes made to the data sets.

A second way is to use PROC CONTENTS after a major data set revision. For example, if you add new variables, drop variables, or re-sort the data, the output from PROC CONTENTS is a useful way to document your activities. As with the original data sets, we recommend both printing the output and saving it to a file.

Tip: A third way is to use PROC COMPARE steps in your programs. This procedure can tell you how the data sets differ, and if they differ in an expected way. The following code uses PROC COMPARE to compare changes between the DEM1 and DEM2 data sets. Both data sets are output data sets from PROC CONTENTS before and after a change to the original data:

```
proc compare data=work.dem1
     compare=work.dem2
     listobs listvar novalues;
   var memname name type length label
       format informat nobs engine
       memtype;
run;
```

This is just one example of using the COMPARE procedure and uses selected options for the procedure. The LISTOBS and LISTVAR options print listings of the observations and variables in one data set only. This is useful when you expect the two data sets to be identical in terms of the number of observations and variables; for example, if your only action has been to sort the data set. The NOVALUES option suppresses the values listing, which can generate a lot of output. The VAR statement identifies what aspects of the data sets to compare.

As an advanced technique, not covered in this paper, you can enclose PROC COMPARE steps in macros. Then, you can check the return code from the procedure, and use PUT statements to write notes in the SAS log.

Use Subsets of Data During Development

As you develop programs to revise the data sets, summarize data, generate graphs, and so on, start with just a subset of the data. This can speed your program development, as you work with smaller data sets. In general, you can often work with 5-10% of the available data during development. With very large data sets, this can significantly reduce run-time for your programs. Once you are fairly certain the program does what you expect, you can begin to work with the full-size data sets.

Tip: In developing your programs, you may eventually use several different subsets to test the program. A useful technique is to create a separate .SAS file which subsets the data appropriately. Then, you can include the subset into your DATA step as shown below:

```
data work.labs;
   set origs146.labs(where=&subset);
run;
```

Consider that you may have several versions of the subset files. As a result, it's best to start with naming conventions that are both useful and informative. Names like SUB1.SAS, SUB2.SAS, and so on inform you of the iterations but not much more. Names like SLABBC.SAS, SLABURN.SAS, SLABLF.SAS are more useful in reminding you that you separated lab results into blood chemistry, urinalysis, and liver function tests.

This is one situation where you do not necessarily want a random sample of the complete data set. For example, suppose your code merges two data sets by PATID to create a third data set. If the first data set has one record per PATID and the second data set has multiple records per PATID, you want to make sure that the same PATIDs are selected in your subset.

Suppose you want to select the first 50 patients from the data set. For data sets with one record per patient, this will be the first 50 records. For data sets with 2 records per patient, this will be the first 100 records, assuming the data set is sorted by patient. This assumption becomes even more important if you want to select the same 50 patients from multiple data sets and use the sample to test merges, concatenations, and so on.

To select a specific subset of records, use the OBS= option. For data set manipulations, use this in a DATA step, as shown below:

```
data work.demog2;
   set cpys146.demog(obs=50);
run;
```

The code above creates a work data set with the first 50 records from the CPYS146.DEMOG data set. Note that the work data set no longer has the label associated with the CPYS146.DEMOG data set. For testing purposes, this is acceptable. If, however, you plan to use the work data set as an interim step to a final data set, you probably want to keep the label.

Another approach is to use the OBS= option with procedures. This can be a good way to test programs that produce reports. The code below uses the OBS= option with PROC PRINT:

```
proc print data=cpys146.demog(obs=50);
   var patid age agerg;
run;
```

This prints a report that uses only the first 50 records in the data set. This approach is most useful with more complex procedures like TABULATE and REPORT.

Consider WRITE or ALTER Passwords

After you have completed work on the data sets, consider whether you want to protect them from accidental overwrite by using a WRITE or ALTER password. SAS provides for three levels of passwords: READ, WRITE, and ALTER. Basically, if you put a READ password on the data file, everyone who wants to access the data file will need to know the password. This can be unwieldy and isn't typical for most purposes. With a WRITE password, you protect the data set from some accidental overwrites. However, the data set can still be replaced by some DATA steps. The examples below use work data sets, but could just as easily use permanent data sets. Consider the following code:

```
data work.demog3(write=secret);
   set work.demog;
run;
proc sort data=work.demog3;
   by age;
run;
data work.demog3;
   set work.demog;
run;
```

The first DATA step places a WRITE password on the data. When running the code interactively, you are prompted for the WRITE password for the PROC SORT step. If you supply an incorrect password, the SORT step won't run. For running in batch, the job runs only if you add the data set option (WRITE=secret) in the PROC SORT statement. The SAS log shows WRITE=XXXXXX so that the password is not revealed.

However, the second DATA step, which re-creates DEMOG3 runs, even if you do not supply a WRITE password. Even with a WRITE password, a DATA step can replace the protected data set. More important perhaps, is that you can delete the protected data set. To provide more stringent control, use an ALTER password as shown below:

```
data work.demog3(alter=secret);
   set work.demog;
run;
proc sort data=work.demog3;
   by age;
run;
data work.demog3;
   set work.demog;
run;
```

In the example above, neither the PROC SORT nor the second DATA step runs. If you run interactively, SAS prompts you for the ALTER password for each step. If you supply the correct password, the step runs; otherwise it doesn't. In batch, you need to add the ALTER= data set option for the code to run. For example, to sort the data in a batch program, use the statements below:

```
proc sort data=work.demog3(alter=secret);
   by age;
run;
```

The SAS log shows ALTER=XXXXXX to keep the password private.

CHECKING DATA SET STRUCTURE

This section covers several techniques for checking the data set structure. Checking structure is an important activity early in the process of working with your data. If you expect the data set to have one record per patient, for example, and this turns out not to be the case, then you may end up throwing away a lot of programming and time.

Checking Data Set Structure

- Sort the data set appropriately, if not sorted.
- Confirm the number of records per-X-per-Y is as expected.
- Investigate potential variables for MERGEs.
- Investigate splitting one large data set into several smaller ones.

Sort Appropriately

First, find out if the data set is sorted. There are three situations to consider: intentionally sorted data (SAS sort flag set), inherently sorted data (ordered but without a sort flag), and unsorted data.

For intentionally sorted data, someone has run a PROC SORT on the data to set the sort flag. The output from PROC CONTENTS can quickly show you the sorting information. As a SAS consultant, it's a good idea to check with the owner of the data (or the programmer who sorted the data) to decide if this sort order is appropriate for what you need to do.

For inherently sorted data, the data set is ordered, but not sorted in the sense that the sort flag is not set. This may be simply a result of chance that the data are ordered in the way they arrived. It's more likely that this is intentional. With small data sets, simply viewing or printing the data set can give you a picture of the inherent sort order. And with small sizes, you can check for errors. With large data sets, this becomes more difficult.

Viewing large data sets becomes less practical and more time-consuming as the size of the data set increases. One useful approach is to determine how you would expect the data to be sorted. Compare your expectations to the data as received. Differences may indicate the presence or absence of variables that identify the structure of the data.

As a SAS consultant, check with the owner of the data. Will it need to be sorted before merging with other data sets? Does a report need to display the information sorted in a given way? If so, you'll find the sort variables.

When SAS sorts a data set, it essentially needs twice the space of the data set for sorting purposes. With large data sets, this can become a problem. As a result, you want to minimize the number of times you need to sort large data sets. Retaining the sorted data set and using it for merges, reports, graphs and other analyses that use BY-variables saves you the frustration of having your program fail due to lack of space or memory.

When you sort, use the OUT= option so that you don't overwrite the original data set.

Another technique with large data sets is to use the NODUP key option in PROC SORT, again using the OUT= option. Large differences between the number of observations in the output data set and your expectations are likely when the structure of the original data set is not as you believe.

Confirm Number of Records Per-X-Per-Y

This section uses pharmaceutical data as an example, but it can extend to many other situations. Often, the data set should have a certain number of records for some identifying variables. In clinical trials data, some data sets should have one record per patient. Other data sets should have one record per patient/visit, or one record per patient/visit/lab test, or one record per patient/visit/adverse event/start date.

When the data are structured and your programs expect the structure to be consistent, you will find problems when the structure is not consistent. In the best case, your program fails and you know right away that a problem exists. In the worst case, your program runs just fine, your QC doesn't catch any unexpected results, but the results still are not correct. These cases are special problems with larger data sets, where complete QC is less likely than with smaller data sets.

A simple way to confirm the number of records for the expected structure is to use PROC SUMMARY to check for uniqueness. Essentially, this approach involves identifying the problem records in the data set and ignoring the records that follow the expected structure.

Consider an example where the data set should have one record for each patient. The statements below create an empty data set if this is true:

```
proc summary nway missing data=cpys146.demog;
   class patid;
   output out=temp1(where=(_freq_>1));
run;
```

Two advantages of using PROC SUMMARY are that it doesn't produce printed output that you don't need, and because of the CLASS statement doesn't require a sort of the data set. The MISSING option tells SUMMARY to consider missing values as a separate class, which can be useful when you do not expect any missing values. In this example, a missing value of PATID, the patient identifier variable, is something you would want to know about and investigate. By using the WHERE clause with _FREQ_>1, the output data set contains only the exceptions. These can then be verified or edited quicker than if you searched the entire data set.

Consider a more complex example where the data set should have one record per patient/visit/adverse event/start date. The statements below create an empty dataset if this is true:

```
proc summary nway data=cpys146.advrs;
   class patid visit aetext aestdt;
   output out=temp1(where=(_freq_>1));
run;
```

With more variables involved in the per-X-per-Y issues and with large data sets, the advantages of the PROC SUMMARY approach are magnified. Again, the data set doesn't need to be sorted, you identify only the problem observations, and you don't get volumes of unneeded output.

Investigate Potential Variables for MERGEs

Checking the per-X-per-Y structure is a first step to investigating the potential variables for MERGEs. The variables that identify the structure of the data are the most likely candidates for merging operations. The reason this is important when checking the structure of large data sets is that you can prevent later sorts of the data. By identifying candidates for MERGEs and sorting by those candidates early, you can avoid re-sorting later.

Investigate Splitting One Large Data Set into Several Smaller Data Sets

For large data sets, it can be particularly useful to split one very large data set into several smaller data sets. This can minimize processing time, as you work with a smaller number of variables and/or records with the smaller data sets.

One situation for splitting occurs when the data set contains commonly-used variables and seldom-used variables. With small data sets, you can just carry along the entire set of variables without much cost in terms of processing time. But, with large data sets, carrying along the unused variables can increase your processing time. Add to this that several procedures (MEANS, for example) automatically perform operations on all numeric variables in a data set. If you have many relatively unimportant variables and inadvertently run PROC MEANS, you may end up with a ream of unnecessary output!

Tip: The code sample below separates the DEMOG data into two data sets that contain commonly-used and seldom-used variables:

```
data cpys146.demg1(label='Common Demographics'
        keep=patid age race sex)
     cpys146.demg2(label='Misc Demographics'
        drop=age race sex);
   set origs146.demog;
run;
```

The code above creates two data sets, one that contains commonly-used variables and one that contains the remaining variables from the large original data set. This code uses a KEEP statement for one data set and a DROP statement for the second. You could use KEEP or DROP statements for either or both data sets. The advantage of using the strategy above is that you ensure the smaller data set keeps just the variables you need, and the other data set drops only the variables that are in the smaller data set. Notice, however, that PATID is not dropped from the DEMG2 data set. This important variable identifies the patients, and is needed for any merge with another data set. When building the DROP list, be sure not to include important identifier variables.

Another case for splitting a data set into multiple data sets occurs when you have a mix-and-match of structures for variables in the data set. For example, suppose some of the variables are collected once per patient and others are collected once per visit. For a small data set, there isn't much cost in carrying along the per-patient variables. For large data sets, we recommend splitting the per-X and per-X-per-Y variables into separate data sets. As an example, separate the per-patient and per-patient-per-visit information into different data sets. The code to split the data sets uses the same strategy as the code shown earlier to split DEMOG into DEMG1 and DEMG2.

CHECKING VARIABLES

This section discusses issues in checking variables in data sets. One overall issue is to check for duplicate information in different variables. With large data sets, and with large numbers of variables in each data set, you want to minimize the situations where duplicate information is maintained in multiple variables or data sets. This can have added benefits of eliminating extra QC and QA steps. For example, suppose you change the format for a variable in one data set. If this same variable exists in other data sets, you need to remember to either use the variable with the new format or to change the variable in the multiple data sets, with QC and QA for each data set that is changed.

Checking Variables

- Identify data redundancies within a data set.
- Determine of one character variable is just a substring of a second character variable.
- Consider using formats for long-valued character variables.
- Consider changing single-digit numeric indicator variables to character variables of length 1.
- Identify duplicate variables in different data sets.

Identify Data Redundancies Within A Data Set

Within a data set, you may find data redundancies, or cases where different variables contain the same information. This is often true if the data sets you receive have undergone pre-processing to create new variables. In these cases, the data may contain extra variables. Particularly with large data sets, you want to drop these unneeded variables. (Remember, you always have a copy of the original data so the deletion of variables occurs in the data sets you work with to produce reports and summaries.)

As an example, the original data may contain separate variables for the day, month, and year of a patient visit. During pre-processing, these variables are combined into an appropriate SAS date value. However, the original variables are retained. For your purposes, these are extra baggage and not needed.

As another example, the original data may contain a variable that records the age of a patient and a separate variable that records age range. You can use just the age variable, and apply a format as needed to generate the age range in reports. This has the added advantage of allowing different age range formats as needed.

The code below illustrates dropping the day, month, year, and age range variables from a data set. Further, it illustrates using AGE with two different formats for age range in two different reports.

```
data cpys146.demog(drop=day month year agerng);
   set origs146.demog;
run;
proc print data=cpys146.demog label;
   id patid;
   var age sex race trt;
   format age rangea.;
run;
proc print data=cpys146.demog label;
   id patid;
   var age sex race trt;
   format age rangeb.;
run;
```

In the example, the RANGEA format separates the ages into categories by groups of 10 (11-20, 21-30, and so on), and the RANGEB format separates the ages into broader categories (21-44, 45-64, >=65). By using formats, the AGE variable is the only variable that needs to be kept in the data set. Otherwise, the data set would contain three variables (AGE, AGERG1, AGERG2) that essentially contained duplicate information.

How do you find these variables? Often, the process of assigning labels to the variables uncovers the redundancies. Or, checking for reasonable values (discussed in “Checking Data Values” later) uncovers potential redundancies. In this example, if you find variables with the values 1-31 and 1-12, you might guess that these are unnecessary day and month variables that can be dropped.

Find Character Substring Variables

If the data set contains character variables, look for situations where one variable is simply a character substring of another variable. One example of this is names of individuals. Suppose NAME contains the entire patient name, FIRST contains only the first name, and LAST contains only the last name. Rather than carry along all three variables, consider keeping only NAME. If needed, you can use the SUBSTR function to get the first and last name for a specific report. If not, you've reduced the size of your large data set by dropping unnecessary variables.

Consider Formats for Character Variables

In some cases, you need to retain the long character values for variables. One common example of this in the pharmaceutical industry is patient or investigator comments. The information written on a case report form must be captured verbatim in the data set. In addition, this is typically diverse enough that using formats would result in a 1-to-1 match between the format and the value.

In many other cases, however, you can save storage space in the data set by using formats for character variables. By using a format and dropping the original variable (from your copy of the original data set), you can decrease the length of the variable.

One example of this from the pharmaceutical industry is signs and symptoms data. Although there may be many values for a character variable, it's likely that those values can be handled with a format. A first way to check for feasibility is to use PROC SUMMARY for a variable. By checking the output (or output data set), you can determine if the number of values can be handled by a format instead of keeping the long character values. If you find that most of the values have a frequency of 1, you may be better off just keeping the long character value. On the other hand, if you find high frequency counts for most values and a few straggler values with only one frequency count, you are better off using formats.

Why not use PROC FREQ? This procedure truncates values at 16 characters. As a result, you may inadvertently group values together—values that should not be grouped into a common format.

Tip: The code below shows how to use PROC SUMMARY to check frequency counts for a variable.

```
proc summary data=cpy146s.signs;
   class symptext;
   output out=temp;
run;
proc print data=temp(where=(_freq_>15));
title 'Symptom Text with More Than 15 Counts';
run;
proc print data=temp(where=(_freq_<=15));
title 'Symptom Text with Less Than 15 Counts';
run;
title;
```

The first PROC PRINT lists the categories of text with more than 15 counts and the second lists the categories of text with fewer than 15 counts. In this example, you want to see most of the categories in the first PRINT listing. Of course, you can adjust the decision rule to fit your needs.

Using Indicator Variables

Suppose you find there are several numeric indicator variables, all of which take on single-digit integer values (between 0 and 9, inclusive). You can convert each of these to character variables with a length of 1 without losing any information. This reduces the size of your data. With large data sets and many indicator variables, the reduction can be significant. For three variables, the information is now stored in 3 bytes instead of 24.

Additionally, as an advanced technique, consider concatenating the character variables into one character variable. By doing so, you don't lose any information but you do reduce the number of variables in your data. Consider the three variables:

OBS	VAR1	VAR2	VAR3	COMBO
1	1	3	7	137
2	9	6	4	964

In the example, you can use the COMBO variable instead of the VAR1, VAR2, and VAR3. By using the SUBSTR function, you can still obtain the original indicator variable values.

Identify Duplicate Variables in Different Data Sets

Typically, you receive multiple data sets for a project. These are likely to have some variables in common—variables you want to use to merge the data sets. However, they may also have variables that simply contain duplicate information. In these cases, consider the following:

- Are you likely to want to merge the data sets that contain duplicate variables? If so, remember that the same-named variables from the second data set will overwrite those in the first data set. To prevent this, you need to rename variables.
- Is one data set of a different, and more appropriate structure? If so, consider keeping the duplicate variable in the appropriate data set and dropping it from the other data sets. With large data sets, you want to keep only the relevant information and keep it in the smallest appropriate data set.

As an example, consider a lab data set from a clinical trial. This data set contains one record per patient/visit/lab test combination. Suppose there are 40 lab tests per visit, 4 visits per patient, and 625 patients. This gives 100,000 observations in the data set (which is actually a fairly small data set by clinical trial standards). Suppose the data set, in addition to lab information, contains the treatment for each patient and that treatment varies on a per-patient basis. In addition, the DEMOG data set for the clinical trial also contains the treatment information.

If you merge the DEMOG and LAB data sets (in that order), the value of TRT from the LAB data set overwrites the value from the DEMOG data set. If you merge LAB and DEMOG, the reverse happens. Are you absolutely certain that the same values are overwriting each other? To answer the question, you need to know what QC was performed to ensure that the TRT value for each patient was the same in the DEMOG and LAB data sets. Magnify this by the fact that the LAB and DEMOG data sets are likely to contain more than one duplicate variable, and your task will take much longer to complete.

A better solution is to perform appropriate QC validation on the values of the variables in the DEMOG data set (one record/patient) and then use the variables from that data set. In building your working copy of the LAB data set, simply use a DROP statement to remove the unneeded duplicate variables from the data set.

CHECKING DATA VALUES

This section discusses strategies for checking data values. For small data sets, you might simply read through a printout of the entire data set. However, this rapidly becomes tedious and is unrealistic for large data sets. For very large data sets, the time to review the data values and the necessary QC is difficult to justify in most data-processing organizations.

Checking Data Values

- Check for reasonable values for variables.
- Check for reasonable values for combinations of variables.
- Find holes, or unexpectedly large numbers of missing values for variables.
- Find holes that are dependent on a per-X-per-variable basis.
- Check for numeric variables with length less than 8.

Check for Reasonable Values for Variables

With small data sets, you can quickly scan the column of values for a variable and find problems. This isn't possible with large data sets due to the labor and time costs alone, much less the eyestrain!

By "reasonable values," we mean checking: minimum and maximum values, nonzero values, missing values, and extreme values. PROC UNIVARIATE can provide you with the printed output containing all of this information. The sample code below can be useful for checking several variables in a data set:

```
proc univariate data=cpy146.labs;
   var rawrslt stdrslt;
run;
```

The UNIVARIATE approach is useful in that you have printed output to review. However, you don't have an output data set. With a large number of variables in many data sets, you are likely to want to build decision criteria to check for reasonable values.

The sample code below uses PROC SUMMARY and decision criteria on the output data set. For the numeric variable AGE, the criteria check for reasonable endpoints for the clinical trial. For the character variable SEX, the criteria check for reasonable character values.

```
proc summary nway missing data=cpys146.demog;
   class age;
   output out=temp(where=(age=. or age<21
          or age>65));
run;
proc print data=work.temp;
run;
proc summary nway missing data=cpys146.demog;
   class sex;
   output out=temp2(where=(sex ^in('MALE'
          'FEMALE')));
run;
proc print data=work.temp2;
run;
```

The output data sets are empty if all values are reasonable (according to the criteria you set). Otherwise, the output data sets contain the values to investigate.

In your industry, you may receive data sets with the same variables. Given this, it may be worth the development effort to build standard checker programs to evaluate reasonable values for variables. As an example, consider demographic information, which is typical of the pharmaceutical industry as well as many other industries. A checker program could identify:

- Ages>100. While possible, these are unlikely enough that they should be checked.
- Negative values for age.
- Values for GENDER other than M/F, Male/Female, or 0/1, depending on the coding strategy.
- Values for RACE other than expected.
- Heights of more than 6'5" or less than 4'6". While possible, these are worth checking. This assumes a study for adults; for children, the ranges would differ.
- Weights outside a defined range. Again, the endpoints for the range will depend on the subjects (adults, children) and the expected values.

The payoff of developing a standard checker program is in re-use of the program. This can become more important with larger data sets, where you can re-use an existing program and complete the check of reasonable values faster than otherwise. Even if you need to make minor changes to the endpoints of ranges, this is likely to give you more time to concentrate on data analysis and reporting.

Check for Reasonable Values of Combinations of Variables

Just as you want to check for reasonable values of individual variables, you can check for reasonable values of combinations of variables.

The variables you use to form combinations and the reasonable values you use are likely to vary with your industry. As an example from the pharmaceutical industry, consider height/weight combinations. You could build a checker program that checks for reasonable combinations of these variables. The criteria are likely to be dependent on the subjects (children or adults) and the medical condition studied. In any case, you want to find subjects with unlikely combinations of values—people over 6'5" who weigh under 100 pounds, for example.

Finding Holes for a Variable

We define *holes* for a variable as being an unexpectedly large number of missing or identical values. This typically indicates problems in the data set—problems that are best discovered early in the process of working with your data. With small data sets, the holes are often easy to find from printouts or reports. With large data sets, you typically do not look through a listing, so the holes are best discovered programmatically.

Consider an example from a data set that contains records for 35,000 patients. The data set contains 42 variables, and you want to identify situations where the number of missing values is unreasonably high. You think the data set has one record per patient, and decide that if a variable has more than 50 missing values, you want to investigate further.

Tip: One way to find holes is to check the number of missing records in the output from SUMMARY or UNIVARIATE. The advantage of using SUMMARY is that you can check an output data set rather than reviewing the printed listings. The following code generates the summary, checks the output data set for holes, and prints the information needed for further investigation.

```
proc summary nway missing data=cpys146.advrs;
   class bodysys;
   output out=temp(where=(bodysys=. & _type_=1));
run;
proc print data=work.temp;
run;
```

The output data set from PROC SUMMARY contains only the record for missing values for BODYSYS. If the count of missing values (in the _FREQ_ variable) is large, then further investigation is needed. You want to find out if this is a potential hole or whether the missing values are acceptable.

If you want to simultaneously check missing values for several variables, the sample code below shows an approach using PROC MEANS.

```
proc means data=cpys146.lab noprint n nmiss
     min max stderr;
   var bc_nval;
   output out=temp1 n=n nmiss=nm min=min max=max
          stderr=se;
run;
proc means data=cpys146.lab noprint n nmiss
     min max stderr;
   var bc_rslt;
   output out=temp2 n=n nmiss=nm min=min max=max
          stderr=se;
run;
data tempc;
   set temp1 temp2;
   if nm>50;
run;
proc print data=tempc;
run;
```

This example defines a hole as being more than 50 missing values.

For finding holes that involve identical values for all observations, PROC SUMMARY can again be used. In this situation, the WHERE condition for the output data set selects only the case where the frequency count matches the number of records in the data set. The sample code below gives an example:

```
proc summary nway missing data=cpys146.advrs;
   class preftxt;
   output out=temp(where=(_freq_=4256));
run;
proc print data=work.temp;
run;
```

As an advanced technique, you can use a macro variable to record the number of observations in the data set and use the macro variable in the WHERE clause.

Finding Holes for Per-X for a Variable

This is an extension of the previous section. In some cases, the holes for a variable are associated with a pattern. Discovering the pattern is the first step in fixing the problem. For example, the holes associated with a variable may be for all the patients at a given clinical site, with the underlying cause being that the information simply wasn't recorded at that site.

Check for Short Numeric Variables

In many cases, SAS data sets have been created from a relational database management system (DBMS). We have found cases where the DBMS stores numeric variables in less than 8 bytes. This is more likely to occur with large tables (and resulting large data sets) because of the perceived savings in storage space. When the SAS data sets are created, this appears to work without problems. However, when the SAS data sets are transported across hosts, problems may occur in that the lengths of these "short numerics" are not the same.

Use PROC CONTENTS to check for numeric variables that are less than 8 in length. The example below uses work data sets but could also use permanent data sets.

```
proc contents data=work.demg noprint
   out=work.tempcon(where=(type=1 & length<8));
run;
quit;

proc print data=tempcon;
   var name type length;
run;
```

The TEMPCON data set contains only those records for numeric variables (TYPE=1) that are less than 8 in length. PROC PRINT provides only the relevant information. By omitting the VAR statement, you can view all the information for these records.

As an additional tip, you can use the same approach to test for long character variables. By changing the WHERE clause to (WHERE=(TYPE=2 & LENGTH>65)), the output data set contains only the long character variables.

In addition, the transporting issues can be solved with the EXTENDSN=NO option in CIMPORT. For example, the following code imports a transport file from UNIX, where the data set associated with the transport file includes numeric variables less than 8 bytes.

```
proc cimport
   infile='c:\miscsas\imprts\lab146.xpt'
   data=orig146.labs extendsn=no;
run;
```

Finally, consider the values for the short numeric variables. A numeric variable of length 4 can handle integer values of up to 256 accurately. Non-integer values or larger values are not necessarily handled accurately. Use the techniques described earlier for checking for reasonable minimum and maximum values to ensure that the values of short numeric variables are handled appropriately.

SUMMARY

When you receive unfamiliar data, your first task is to learn about the data. For small files, you may be able to simply look at a listing of the data. As files become larger, this becomes less feasible. We have presented a number of tips and techniques that have helped us efficiently learn about large data sets. Some of the tips are just good programming practice (documenting data set changes, for example). Other tips apply to data sets of all sizes (using CONTENTS to document data sets, for example). Others are most useful with large data sets (confirming the number of records per-X-per-Y, for example). All of the techniques become more useful and more important for large data sets. The effort to manually check data set issues and perform the QC to confirm the results becomes virtually impossible with large data sets. Learning about your data using programs and their results is the best approach with large data sets.

AUTHOR CONTACT INFO

Sandra D. Schlotzhauer
Schlotzhauer Consulting
400 Gibbon Drive
Chapel Hill, NC 27516
919-933-6341
sdschlotz@aol.com

Bob Anschuetz
Quintiles Inc
P.O. Box 13979
919-941-7135
Research Triangle Park NC 27709
banschue@quintiles.com