MWSUG 2011 - Paper S111

Dealing with Duplicates in Your Data

Joshua M. Horstman, First Phase Consulting, Inc., Indianapolis IN
Roger D. Muller, First Phase Consulting, Inc., Carmel IN

Abstract

As SAS programmers, we deal with data all the time. We understand the principle of "garbage in, garbage out." Data is an essential component of the modern business, and erroneous data can have serious ramifications. In today's world of globally distributed teams, we often work with data coming from sources far beyond our control. This makes it even more critical for the SAS programmer to become adept at dealing with data issues.

In this paper, we address one of the most common data issues: duplicate data. Using simple examples, we describe different types of duplicate data and describe strategies for identifying questionable records so they can be evaluated and appropriate action taken. This requires the programmer to understand which variables in their data constitute a unique key (BY variables). The paper also features an in-depth discussion of options on PROC SORT (NODUP, NODUPKEY, DUPOUT) as well as alternative methods such as PROC FREQ and PROC SQL. This paper is appropriate for users with any level of SAS programming expertise.

Introduction: What is the Most Important Statistic in a Dataset?

We would contend that the most important statistic in a dataset is not the mean, the median, nor any measure of variation. Rather, it is **N --- the number of observations**. If a dataset does not contain the number of observations that it should, any analysis and reporting performed on the data are suspect. The principle of "garbage in, garbage out" applies here, and the consequences can be far-reaching and potentially devastating to a business.

There are many reasons why a dataset might not have the correct number of observations, but duplicate data is a very common one. In this paper, we'll discuss issues related to duplicate data and ways to use SAS to detect and remove duplicate data.

Defining Duplicate Data

Before we begin the discussion of duplicate data, it is important to clearly define the problem we are attempting to address. When some speak of "duplicate data", they refer to a situation in which multiple copies of data exist scattered around on various systems and in various versions. This is a real problem that could be discussed at length, but it is beyond the scope of this paper. Here, we are concerned with duplicate records (also called duplicate observations or duplicate rows) that exist within a single dataset.

Getting to Know Your Data

First, be honest with yourself. If you are not a subject matter expert in the area you are summarizing, do not hesitate to ask someone who is. Throw it back over the fence! The database manager may not be a subject matter expert either (in fact, they seldom are). The true client (scientist, engineer, marketer, administrator, etc.) needs to be aware of questions in the dataset being summarized. Here are some techniques for getting to know your data:

Printouts:
Simply dumping the data or portions of it to a print file for online examination or to paper can help you understand it better.

Interactive Examination:
Use the Viewtable facility in interactive SAS or the tools for query and sorting in SAS Enterprise Guide (version 4.2 or higher).

PROC FREQ: Very useful for gaining an overview of alphanumeric data. Look carefully at the N value counts to see what they are and whether they agree with what you expect.

PROC UNIVARIATE: Very useful for gaining an overview of numeric information. Again, make sure N agrees with your expectations. Look at the frequency distribution. Look at the number of zeroes and missing values. The improper recording of missing numeric values as zero values is a frequent mistake and can cause unexpected results. As a matter of standard SAS practice, missing numeric values are represented as a period, which has a different meaning than a zero.

Graphics Procedures: Visualizing data can be a very effective way to easily spot errors, duplicates, and other anomalies. SAS/GRAPH includes several procedures that can be used to quickly create a visual representation of your data.

Are Duplicate Records Always Wrong?

Absolutely not. Nor are they always right. You have to be enough of a subject matter expert to make that determination. Consider the following dataset containing bowling scores:

BOWLER	SCORE
Josh	232
Roger	112
James	195
Roger	112

Notice that there are two rows containing Roger's name with a score of 112. While this is a duplicate record, we cannot say that it is wrong without more information on the subject. After all, there could have been two bowlers named Roger who just happened to have the same score. Alternatively, Roger could have bowled twice. Without knowledge of the nature of the data and how it was collected, we are unable to say for certain.

This highlights the benefit of establishing a unique key for your data. A unique key is a set or variables whose combination of values are always unique by definition for each row. In this example, we could have added a bowler identification number to eliminate the possibility of confusing two different bowlers with the same name. Additionally, we could have added a game identification number to make it clear if someone bowled multiple games.

Another option is to make sure records have a date and time. If a medical patient has 2 entries for a given lab test with exactly the same results and the records have the same date and time, it is highly likely that this is an illegitimate duplicate record. On the other hand, if the dataset contains only a date and not time, what are you to conclude? Is it possible that samples were taken at different times during the day and everything was the same? Could the record be legitimate? Possibly. It is time for you to "throw this one back over the fence".

Detecting and Removing Duplicates

Now that we are familiar with our data and we have determined what constitutes invalid duplicate data, we turn our attention to the specific methods for detecting and removing duplicates.

As always, it pays to read your SAS log file. Many data problems can be uncovered by observing the observation counts and investigating the notes, warnings, and errors in the log. In particular, watch the log for "NOTE: MERGE statement has more than one data set with repeats of BY values." While there are a few circumstances where this is unavoidable, it usually is a strong clue that all is not well.

Beyond checking the log, there are several methods to use SAS to proactively search for invalid duplicates in your data. We will proceed to describe four such methods.

Our Sample Dataset

Before we proceed to describe specific methods for detecting and removing duplicates, we describe a sample dataset that will be useful in illustrating the operation of these methods. This dataset contains orders for cakes at a bakery as shown in Table 1. Each record includes an order identification number as well as three variables that provide specifications for the order. The order identification number is intended to be a unique key useful for identifying a particular order.

It should be noted that two different data issues have been deliberately incorporated into our sample dataset so that we might explore how the different methods deal with them. First, a duplicate entry for order #002 has been included in the dataset. The two rows are completely identical. Secondly, #004 has been entered in triplicate. The first and third rows for order #004 are identical, while the one in between differs in some of the order attributes.

Table 1. Original CAKE_ORDERS Dataset

ORDER_ID	SIZE	SHAPE	FLAVOR
001	LARGE	ROUND	CHOCOLATE
002	SMALL	ROUND	YELLOW
002	SMALL	ROUND	YELLOW
003	LARGE	RECTANGLE	WHITE
004	LARGE	SQUARE	CARROT
004	LARGE	ROUND	YELLOW
004	LARGE	SQUARE	CARROT
005	SMALL	HEART	RED_VELVET

Method 1: PROC SORT

Perhaps the most well-known method of detecting and removing duplicates in SAS is using the SORT procedure. The PROC SORT statement has three options that are particularly useful in dealing with duplicates: NODUPRECS, NODUPKEYS, and DUPOUT=. We will discuss each one in turn and show how it would work with our sample dataset.

The NODUPRECS (or NODUP) option

When the SORT procedure is used with the NODUPRECS option (or its alias NODUP), duplicate records are automatically removed from the output dataset. It is important to note that only duplicate records which have identical values for all variables will be removed. Thus, we should not expect that this procedure will eliminate the second row for order #004 since it has different values for SHAPE and FLAVOR than the first row for that order. However, we might expect that the second row for order #002 and the third row for order #004 will be removed since they are each identical to other rows.

Since the SORT procedure always requires at least one BY variable, here we use our ORDER_ID:

```
proc sort data=cake_orders noduprecs; by order_id; run;
```

Table 2 displays the resulting output dataset. Careful inspection reveals that this is not the expected output. While the duplicate record for #002 was removed, there are still three rows corresponding to order #004.

Table 2. CAKE_ORDERS after PROC SORT by ORDER_ID with NODUPRECS

ORDER_ID	SIZE	SHAPE	FLAVOR
001	LARGE	ROUND	CHOCOLATE
002	SMALL	ROUND	YELLOW
003	LARGE	RECTANGLE	WHITE
004	LARGE	SQUARE	CARROT
004	LARGE	ROUND	YELLOW
004	LARGE	SQUARE	CARROT
005	SMALL	HEART	RED_VELVET

How can this be explained? Why did the NODUPRECS option fail to remove the third row for order #004? The answer can be found in a deeper understanding of how the NODUPRECS option actually works. As the dataset is sorted, each record being written to the output dataset is compared with the previous record. If it is identical to the previous record, it is not written. No consideration is given to nonconsecutive duplicates.

In the case of our sample dataset, the three rows for order #004 were already sorted based on the BY variables specified (only ORDER_ID), so they would be written to the output dataset in the same order they appeared in the original dataset. Since the two identical rows were nonconsecutive, the NODUPRECS option failed to remove one of them.

In the experience of the authors, this peculiar behavior of the NODUPRECS option has gone unnoticed by many SAS programmers because it only affects the results under relatively uncommon circumstances. This example was carefully constructed to expose this failure. Nonetheless, this result demonstrates the real risk taken by the programmer who employs the NODUPRECS option carelessly.

Fortunately, there is a simple way to prevent this problem. By adding additional variables to the BY statement, the occurrence of nonconsecutive duplicates can be reduced. By adding all variables to the BY statement, it can be eliminated entirely. The latter is the practice recommended by the authors, and it can be easily implemented using the _ALL_ keyword as follows:

```
proc sort data=cake_orders noduprecs; by _ALL_; run;
```

Including all variables in the BY statement forces identical records into consecutive positions where they can be successfully eliminated by the NODUPRECS option. (It is worth noting that this could become quite resource-intensive if there are a large number of variables in the dataset.) The results are shown in Table 3.

Table 3. CAKE_ORDERS after PROC SORT by _ALL_ with NODUPRECS

ORDER_ID	SIZE	SHAPE	FLAVOR
001	LARGE	ROUND	CHOCOLATE
002	SMALL	ROUND	YELLOW
003	LARGE	RECTANGLE	WHITE
004	LARGE	ROUND	YELLOW
004	LARGE	SQUARE	CARROT
005	SMALL	HEART	RED_VELVET

The NODUPKEY option

While the NODUPRECS option is useful for removing exact duplicate records, it is often the case that we wish to broaden our definition of duplicate data. Frequently, there is a subset of variables that should constitute a unique key for our data. When that uniqueness constraint is violated, we have a data issue that we must address. In the case of our example dataset, each order is supposed to have a unique order identification number. However, even after removing exact duplicates, we still have two rows with an ORDER_ID value of 004.

This is where the NODUPKEY option of the SORT procedure comes into play. When the SORT procedure finds multiple records with the same values for the BY variables, only the first such record is written to the output dataset. In our example, our unique key consists of only the ORDER_ID variable, so this is only variable we include in the BY statement:

```
proc sort data=cake_orders nodupkey; by order_id; run;
```

The output is shown below in Table 4. There are now only five rows remaining, one for each of the five unique values of ORDER_ID.

Table 4. CAKE_ORDERS after PROC SORT by ORDER_ID with NODUPKEY

ORDER_ID	SIZE	SHAPE	FLAVOR
001	LARGE	ROUND	CHOCOLATE
002	SMALL	ROUND	YELLOW
003	LARGE	RECTANGLE	WHITE
004	LARGE	SQUARE	CARROT
005	SMALL	HEART	RED_VELVET

It is tempting to think that we have now sufficiently dealt with our duplicate data. However, it is important to note one key fact about our use of the NODUPKEY option: we have lost information. Our original dataset contained two rows for order #004 with different specifications for the order. When we ran the SORT procedure with the NODUPKEY option, we simply kept the first one and dropped the second one.

However, this may not be correct. We don't know which row provides the correct specifications for the order. It's also possible that these rows represent two distinct orders which never should have been assigned the same order identification number in the first place. This is not a determination that can be made based solely on the original dataset. We will need to seek additional information from the source of our data to determine what has gone wrong.

As an aside, this situation also highlights one of the broader dangers of using the NODUPKEY option. This option is often used by programmers to obtain a list of unique values for a variable. However, when the NODUPKEY option is used in this way, it is usually advisable to drop all other variables from the output dataset as they now contain only an

arbitrary subset of the original data. If these other variables are brought along for the ride, this extra baggage may subsequently produce erroneous results if used improperly.

The DUPOUT= option

We have identified the need to seek additional information from an external source so that we can determine the appropriate way to deal with the duplicate data. Before we can make such an inquiry, we need to identify the problematic rows from the original dataset. In our example dataset, this information can be easily determined by visual inspection. However, since many business scenarios involve vastly larger quantities of data, it is useful to automate this process.

The SORT procedure includes a DUPOUT= option which can be helpful here. This option is used to specify an output dataset to which duplicate observations will be written. We can make use of this option to see which observations are removed from our dataset by the NODUPRECS or NODUPKEY options.

```
proc sort data=cake_orders nodupkey dupout=cake_dups; by order_id; run;
```

Table 5 displays the output dataset created by the DUPOUT option. Note that this dataset is separate from the normal output dataset that the SORT procedure creates. The procedure generates both at once.

Table 5. CAKE_DUPS created by DUPOUT option of PROC SORT with NODUPKEYS

ORDER_ID	SIZE	SHAPE	FLAVOR
002	SMALL	ROUND	YELLOW
004	LARGE	ROUND	YELLOW
004	LARGE	SQUARE	CARROT

Now we have a list of only those observations which were dropped from our dataset by the SORT procedure. However, the presentation of this information is less than ideal. For a pair of observations of duplicate keys, the first observation will be in the OUT= dataset and the duplicate will be in the DUPOUT= dataset. As we mentioned earlier, in the case of duplicate keys, it is not always clear which of these is the “correct” observation, and it would be nice to compare them together. In order to accomplish this, we will have to move beyond a simple PROC SORT.

Method 2: DATA Step with FIRST. Or LAST.

A more convenient listing of rows with duplicate keys can be obtained using a simple two-step process. The first step is to call PROC SORT to sort the data by the relevant key variables. The second step is to use a DATA step to output only those rows which do not correspond to a unique set of values for those key variables. If a row corresponds to a unique set of values for the key variables then both FIRST.varname and LAST.varname will be true, where varname is the name of the last BY variable. Thus, we proceed by keeping only those rows where at least one of those two expressions does not evaluate to true:

```
proc sort data=cake_orders; by order_id; run;

data cake_dups;
  set cake_orders;
  by order_id;
  if not (first.order_id and last.order_id);
  run;
```

The resulting dataset is shown below in Table 6. The output includes both of the duplicate records for order #002 as well as all three of the records for order #004. We now have a single dataset that includes all of the duplicate records.

This information will be of interest to someone who is responsible for investigating and resolving these data issues. However, this could become unwieldy for a large dataset. We will look at other methods that generate additional output that may be of interest.

Table 6. CAKE_DUPS created by DATA Step using FIRST. And LAST.

ORDER_ID	SIZE	SHAPE	FLAVOR
002	SMALL	ROUND	YELLOW
002	SMALL	ROUND	YELLOW
004	LARGE	SQUARE	CARROT
004	LARGE	ROUND	YELLOW
004	LARGE	SQUARE	CARROT

Method 3: PROC FREQ

Another way of detecting duplicate records in a dataset is by generating a frequency count for each set of values of the key variables. This can be accomplished using the FREQ procedure. Here we suppress the printed output and generate an output dataset. We drop frequency counts equal to 1 since we are interested here in those values that occur more than once.

```
proc freq data=cake_orders noprint;
  table order_id / out=cake_dupcount(where=(count>1) drop=percent);
  run;
```

The TABLE statement should include all key variables, separated by asterisks. In our example, there is only one key variable. Note that if we were interested in counting exact duplicates, we would need to include all variables on the TABLE statement. Unfortunately, this cannot be accomplished by including the _ALL_ keyword on the TABLE statement.

Table 7 shows the output dataset created by the FREQ procedure. This provides a nice summary of the duplicate data issues that need to be addressed in our dataset.

Table 7. CAKE_DUPCOUNT created by PROC FREQ

ORDER_ID	COUNT
002	2
004	3

Method 4: PROC SQL

The FREQ procedure is not the only way to produce such a summary. An alternative method involves the use of PROC SQL as follows:

```
proc sql;
  create table cake_dupcount as
    select order_id, count(order_id) as count
      from cake_orders
      group by order_id
      having count > 1;
  quit;
  run;
```

Table 8. CAKE_DUPCOUNT created by PROC SQL grouped by ORDER_ID

ORDER_ID	COUNT
002	2
004	3

Note from Table 8 above that the output is identical to what we obtained using the PROC FREQ method above. However, there are a couple of potential advantages to this method. First of all, PROC SQL is more efficient in certain situations than PROC FREQ, so it may be a better option when dealing with large datasets or datasets with a large set of key variables.

Secondly, if we are interested in counting exact duplicate rows (as opposed to just certain key variables), this can be done without the need to explicitly specify the names of all of the variables in the dataset. The code below shows one way this can be accomplished with the output following in Table 9.

```
proc sql noprint;
  select name
    into :colnames separated by ', '
    from dictionary.columns
    where libname="WORK" and memname="CAKE_ORDERS";
  create table cake_dupcount as
    select &colnames, count(*) as count
    from cake_orders
    group by &colnames
    having count > 1;
  quit;
  run;
```

Table 9. CAKE_DUPS created by PROC SQL grouped by all columns

ORDER_ID	SIZE	SHAPE	FLAVOR	COUNT
002	SMALL	ROUND	YELLOW	2
004	LARGE	SQUARE	CARROT	2

Conclusion

The SAS programmer has an extensive collection of tools available to find and correct problems with a dataset. In order to most effectively use these tools, it is helpful for the programmer to understand the broader context of the data. Know your data; make sure that the data you think you have is the data you actually have; remember that N is the most important statistic for a dataset; and don't hesitate to return to the source of your data when things aren't right to seek additional information, clarification, or corrections.

Recommended Reading

Franklin, David. "Finding Duplicate Records in a SAS Dataset." Pharmaceutical SAS Users Group 2007 Proceedings, Paper PO14. http://www.lexjansen.com/pharmasug/2007/po/po14.pdf

Peterson, Brett J. "Finding a Duplicate in a Haystack." SUGI 31 Proceedings, Paper 164-31, 2006. http://www2.sas.com/proceedings/sugi31/164-31.pdf

Wright, Wendi L. "Checking for Duplicates." SAS Global Forum 2007 Proceedings, Paper 069-2007. http://www2.sas.com/proceedings/forum2007/069-2007.pdf

Contact Information

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

Roger D. Muller
First Phase Consulting, Inc.
317-846-5782
rdmuller@hotmail.com

Joshua M. Horstman
First Phase Consulting, Inc.
317-815-5899
jmhorstman@gmail.com