Paper S1-14-2012

Making it Add Up; Diagnosing Merge Issues Using PROC SQL and Simple Arithmetic

Eric Schreiber, PRA, Lenexa, KS

ABSTRACT

Data set merges are an everyday part of working in SAS. Often times these merges are simple and easy, especially while merging subsets from a singular source. Merging data sets from different sources can be a bit trickier, however. One missing letter can be the difference between a clean merge and accidentally removing valuable data. Picking through the non-matching data, character by character, can be a long and tedious process but there are ways to programmatically find these differences; one of which will be explained in this paper.

INTRODUCTION: WHY WORRY ABOUT MERGING INCONSISTENCIES?

In the pharmaceutical industry, we deal with merges using differing source data sets on a regular basis. Many times, for one reason or another, we want to keep only the records from one set or the other. This does well to account for the time difference between data sources but it also yields an undesirable effect. We end up throwing out records because of something as simple as a missing character. This is unacceptable and needs to be remedied, but is there a quicker way than doing a visual check? The answer is, of course, yes, but before we delve into a programmatic check for merge issues let us take the time to understand a key component of the process.

A QUICK NOTE ABOUT PROC SQL

PROC SQL can be very helpful in uncovering variable differences between data sets. In order to understand the process, however, it is important to know how the GROUP BY function in PROC SQL works. We will use the following data set, table 1, to help us illustrate:

SUBJECT	RECORD	VAR
XYZ-001-0101	a	1
XYZ-001-0101	b	1
XYZ-001-0102	a	1
XYZ-001-0102	b	2

Table 1. Data set DS1

To put it simply, GROUP BY will allow us to separate our defined functions into smaller segments or by groups. For example, let's say we want to sum the numeric variable *VAR*. A PROC SQL statement that does not use GROUP BY will sum *VAR* over the entire data set. However, if we add GROUP BY, PROC SQL will now sum *VAR* across each subset specified in the GROUP BY statement:

```
Proc sql noprint;
    Create table DS2 as
    Select subject, record, sum(var) as count
    From DS1
    Group by subject;
Quit;
```

Using the above PROC SQL code without the GROUP BY statement we will return this result:

SUBJECT	RECORD	VAR	COUNT
XYZ-001-0101	a	1	5
XYZ-001-0101	b	1	5
XYZ-001-0102	a	1	5
XYZ-001-0102	b	2	5

Table 2. Data set DS2 (without GROUP BY subject)

If we now add GROUP BY *SUBJECT*, our summing will look different:

SUBJECT	RECORD	VAR	COUNT
XYZ-001-0101	a	1	2
XYZ-001-0101	b	1	2
XYZ-001-0102	a	1	3
XYZ-001-0102	b	2	3

Table 3. Data set DS2 (with GROUP BY subject)

So, as you can see, when using sum without a GROUP BY statement you will get a summation of the numeric variable over the entire data set (1+1+1+2). However, when GROUP BY is added, the summation will occur over each individual by group (1+1 for subject 0101 and 1+2 for subject 0102). With a clear understanding of GROUP BY we can now move forward and apply this to diagnosing merge issues.

DIAGNOSING MERGE ISSUES

ESTABLISHING A TARGET DATA SET

Before we can find merge issues we need at least two data sets. For our purposes we will use exactly two data sets, APPLES (table 4) and ORANGES (table 5).

SUBJECT	DATE	TIMEPOINT	CATEGORY
XYZ-001-0101	2012-01-01T10:15:00	1 hours after dose	Chemistry
XYZ-001-0101	2012-01-01T12:15:00	3 hours after dose	Chemistry

Table 4. Data set APPLES

SUBJECT	DATE	TIMEPOINT	CATEGORY	TEST	RESULT
XYZ-001-0100	2012-01-01T10:15:00	1 hours after dose	Chemistry	test	10
XYZ-001-0101	2012-01-01T10:14:00	1 hours after dose	Chemistry	test	12
XYZ-001-0101	2012-01-01T10:15:00	1 hours after dose	Chemistry	test	12
XYZ-001-0101	2012-01-01T12:15:00	3 hour after dose	Chemistry	test1	8
XYZ-001-0101	2012-01-01T12:15:00	3 hours after dose	Chemistry	test1	9
XYZ-001-0101	2012-01-01T12:15:00	3 hours after dose	Chemistry	test1	12

Table 5. Data set ORANGES

Although we could look at both together, it is easier to see the merge problems if we establish a target data set. We will use data set ORANGES as our target. This means that we will determine which records from ORANGES are not merging properly into APPLES and which variables are causing the problem.

OBTAIN A UNIQUE SET OF RECORDS BY MERGE KEY

Now that we have established a target data set we can begin the check process. Because we are only concerned with each unique combination of our sort key, we can drop all variables that are not part of the sort key and can then delete all duplicates using a NODUPKEY or any other preferred method.

```
Proc sort data=oranges(keep=subject date timepoint category) out=oranges1
                nodupkey;
    by subject date timepoint category;
Run;
```

The same deletion method as described above can be used on data set APPLES. Because there are no exact duplicates in APPLES and because only the sort key variables are present in the data, no records are altered.

SUBJECT	DATE	TIMEPOINT	CATEGORY
XYZ-001-0100	2012-01-01T10:15:00	1 hours after dose	Chemistry
XYZ-001-0101	2012-01-01T10:14:00	1 hours after dose	Chemistry
XYZ-001-0101	2012-01-01T10:15:00	1 hours after dose	Chemistry
XYZ-001-0101	2012-01-01T12:15:00	3 hour after dose	Chemistry
XYZ-001-0101	2012-01-01T12:15:00	3 hours after dose	Chemistry

Table 6. Data set ORANGES1

As you can see in table 6, the variables *TEST* and *RESULT* have been dropped from our ORANGES data sets. We have also omitted 1 duplicate record as both they contain the same sort key. This consolidated record is highlighted in blue.

Although this may seem like a minor detail, the deletion strategy will add much flexibility to our merge check. We are now able to check all merge types by keeping only unique records. This means that one-to-one, one-to-many, many-to-one and many-to-many merges will work.

MERGE APPLES AND ORANGES

At this point we can begin the merge process. We will now merge ORANGES1 and APPLES1 by our specified sort key, keeping only the records from ORANGES1 that do not merge with the APPLES1 data set. These will be the records that do not match on our merge key and need further evaluation. To accomplish this we can use a simple DATA step:

```
Data oranges_nonmerge;
    Merge oranges1(in=oranges) apples1(in=apples);
    By subject date timepoint category;
    If oranges and not apples;
Run;
```

By using the IN= option in our above DATA step we are able to obtain the records that are in ORANGES1 but not in APPLES1. Table 7 reveals three records from dataset ORANGES1 that do not merge with APPLES1. This means that, for each record in this table, one of the four variables does not match what we have in APPLES1.

SUBJECT	DATE	TIMEPOINT	CATEGORY
XYZ-001-0100	2012-01-01T10:15:00	1 hours after dose	Chemistry
XYZ-001-0101	2012-01-01T10:14:00	1 hours after dose	Chemistry
XYZ-001-0101	2012-01-01T12:15:00	3 hour after dose	Chemistry

Table 7. Data set ORANGES_NONMERGE

Once our non-merges are identified we will need to create an identifier variable and a numeric summing variable to assist with the rest of the process. In a simple DATA step we can create an identifier variable named *DS* and set it to 'apples' for the APPLES1 data set and set to 'oranges' for the ORANGES_NONMERGE data set. We will also create a variable called *NUM* and set it to 1 in ORANGES_NONMERGE and 1000 in APPLES1.*

*Depending on the number of records in your non-merge data set, the number assigned to the non-target dataset needs to be a significantly higher than the number assigned to the non-merge data set in order to avoid potentially incorrect results from this test. A ratio of 1:(X+1), where X is the number of records in the non-merge data set, is recommended.

```
Data oranges_nonmerge2;
       Set oranges_nonmerge;

       Ds='oranges';
       Num=1;
   Run;

   Data apples2;
       Set apples1;

       Ds='apples';
       Num=1000;
   Run;
```

After assigning these new variables we will set ORANGES_NONMERGE2 and APPLES2 together to procure the data set we will use to find our mismatching columns.

```
   Data apples_oranges;
       Set oranges_nonmerge2 apples2;
   Run;
```

SUM NUMERIC VARIABLE TO UNCOVER MISMATCHED VARIABLE

At this point we are ready to begin identifying our mismatching variables using the GROUP BY feature and vertical finesse of PROC SQL. We take our recently created data set, comprised of the non-merges from ORANGES and the entire APPLES data set, and sum the variable *NUM*, grouping by the first variable in our sort key:

```
   Proc sql noprint;
       Create table merg as
       Select *, sum(num) as count
       From apples_oranges
       Group by subject;
   Quit;
```

Giving us the following data set:

SUBJECT	DATE	TIMEPOINT	CATEGORY	DS	NUM	COUNT
XYZ-001-0100	2012-01-01T10:15:00	1 hours after dose	Chemistry	oranges	1	1
XYZ-001-0101	2012-01-01T10:14:00	1 hours after dose	Chemistry	oranges	1	2002
XYZ-001-0101	2012-01-01T10:15:00	1 hours after dose	Chemistry	apples	1000	2002
XYZ-001-0101	2012-01-01T12:15:00	3 hour after dose	Chemistry	oranges	1	2002
XYZ-001-0101	2012-01-01T12:15:00	3 hours after dose	Chemistry	apples	1000	2002

Table 8. Data set MERG

Let us take a close look at the *COUNT* variable in table 8. After summing *NUM* and grouping by *SUBJECT* we see counts of 1 and 2002. Because our first record has a subject number different from the rest and because it is from dataset ORANGES, its *COUNT* will only be 1 (the sum of 1 is 1). Records 2 through 5 all have a similar subject number and so the sum of these records is 2002 (1+1000+1+1000).

From this we can see that any variable with a count of less than 1000 does not match with the APPLES data set on variable SUBJECT. If it did have a match the value of 1000 assigned to the records from APPLES would push the summed *COUNT* variable up over 1000. So again, when count is less than 1000, we can create a new text variable stating that this specific record does not match on the *SUBJECT* variable:

```
Proc sql noprint;
    Create table tt1 as
    Select *, case
                When count lt 1000 then "No match in variable subject"
                Else ''
                End as text1
    From merg;
Quit;
```

SUBJECT	DATE	TIMEPOINT	CATEGORY	DS	NUM	COUNT	TEXT1
XYZ-001-0100	2012-01-01T10:15:00	1 hours after dose	Chemistry	oranges	1	1	No match in variable subject
XYZ-001-0101	2012-01-01T10:14:00	1 hours after dose	Chemistry	oranges	1	2002	
XYZ-001-0101	2012-01-01T10:15:00	1 hours after dose	Chemistry	apples	1000	2002	
XYZ-001-0101	2012-01-01T12:15:00	3 hour after dose	Chemistry	oranges	1	2002	
XYZ-001-0101	2012-01-01T12:15:00	3 hours after dose	Chemistry	apples	1000	2002	

Table 9. Data set TT1

REPEAT AND RE-MERGE

Our next test will check the second variable in our sort key and so we use similar code but this time we read-in the previously created data set and GROUP BY *SUBJECT* and *DATE*:

```
Proc sql noprint;
    Create table merg1 as
    Select sum(num) as count
    From tt1
    Group by subject, date;
Quit;
```

And, again, we come out with a similarly structured data set:

SUBJECT	DATE	TIMEPOINT	CATEGORY	DS	NUM	TEXT1	COUNT
XYZ-001-0100	2012-01-01T10:15:00	1 hours after dose	Chemistry	oranges	1	No match in variable subject	1
XYZ-001-0101	2012-01-01T10:14:00	1 hours after dose	Chemistry	oranges	1		1
XYZ-001-0101	2012-01-01T10:15:00	1 hours after dose	Chemistry	apples	1000		1000
XYZ-001-0101	2012-01-01T12:15:00	3 hours after dose	Chemistry	apples	1000		1001
XYZ-001-0101	2012-01-01T12:15:00	3 hour after dose	Chemistry	oranges	1		1001

Table 10. Data set MERG1

As before, any variable that has a count of less than 1000 is a mismatch. We also need to add an additional condition for variable *DATE*; we need to check whether or not our text variable is already occupied. If a record does not match on *SUBJECT* then of course it will not match on *SUBJECT* and *DATE*. However, if it did match on *SUBJECT* and we now have a count of less than 1000 then we can mark the *DATE* variable as the problem maker.

This same process can be repeated for each of the remaining variables in the sort key, each time checking to see if *COUNT* is less than 1000 and if the text variable has already been populated.

Once all variables have been checked and our text variables have been established we can output just the non-merges from the ORANGES data set (where *DS*='oranges'). After sorting by the sort key, these can be remerged with the original ORANGES data set. Because we keep the text variable, we are able to see exactly which records from ORANGES do not merge into APPLES and exactly which variable in the merge key is the problem.

SUBJECT	DATE	TIMEPOINT	CATEGORY	TEST	RESULT	TEXT
XYZ-001-0100	2012-01-01T10:15:00	1 hours after dose	Chemistry	test	10	No match in variable subject
XYZ-001-0101	2012-01-01T10:14:00	1 hours after dose	Chemistry	test	12	No match in variable date
XYZ-001-0101	2012-01-01T10:15:00	1 hours after dose	Chemistry	test	12	
XYZ-001-0101	2012-01-01T12:15:00	3 hour after dose	Chemistry	test1	8	No match in variable timepoint
XYZ-001-0101	2012-01-01T12:15:00	3 hours after dose	Chemistry	test1	9	
XYZ-001-0101	2012-01-01T12:15:00	3 hours after dose	Chemistry	test1	12	

Table 11. Data set Final

This final merge is also the reason we are able to delete duplicate records. As discussed above, this allows us to check each possible type of merge.

IDEAS FOR FUTURE DEVELOPMENT

While this strategy does go a long way in helping to identify merge issues, it does have a few limitations. Next we'll discuss two limitations and ways to overcome them.

CHECKING EACH MISMATCHING VARIABLE CHARACTER BY CHARACTER

The merge check from above does identify the problematic variable but it would be even more convenient to locate the exact character that is not matching. Once we have the variable mismatch we are well on our way to checking the character difference as well. Once our mismatching variable is identified we could repeat the above process but use each character of our mismatching variable as the sort key in order to find the exact problematic character.

Although this could potentially require a substantial amount of computing time, it would still be much quicker than looking through the data manually.

SCRAMBLE THE SORT CODE AND CHECK IT AT EACH COMBINATION OF THE DATA

You may be asking yourself, what happens when we have more than 1 mismatching variable? Using the above method only the first mismatch is identified. Assuming a sort key in which more than 1 variable is a mismatch, the order of the variables is the key in determining which variable will be flagged. To check for multiple mismatches we could run the program multiple times, using different sort key combinations and then merge the results at the end. This would, like the character check, require more computing time.

CONCLUSION

Merging data sets from two sources can be a painstaking process but with the right tools it doesn't have to be a game of I Spy. Using the vertical calculating power of PROC SQL and a little bit of simple addition, it is easy to find merge issues in a few quick programming steps.

ACKNOWLEDGEMENTS

I'd like to thank my colleagues Lesley Alagar, Craig Mistal and Andy Hulme for taking the time to review this paper and offer advice.

CONTACT INFORMATION

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

Name: Eric Schreiber
E-mail: ejschreiber1@gmail.com