How MERGE Really Works

Bob Virgile
Robert Virgile Associates, Inc.

Overview

Do your MERGEs produce unexpected results? Three basic DATA step concepts resolve the complexities of MERGE: compile and execute, the Program Data Vector, and data movement among various storage locations. These concepts apply to all DATA steps, not just those using MERGE. This paper examines these key issues in general and then applies them specifically to the MERGE process. Examples illustrate trickier aspects of MERGE as well as techniques to code around the pitfalls.

Three Key Concepts

All SAS® DATA steps employ three key concepts:

1. Compile and execute.[1] The SAS software compiles the statements within the DATA step, and performs any required set-up work for each statement. Then, the software executes the programming statements for each observation.

2. The Program Data Vector (PDV). As part of the compilation process, the software sets up storage locations in memory to hold the current values of all variables.

3. Data movement. As the DATA step executes, values are placed into the PDV and later copied from the PDV to output SAS data sets.

The details of the internal workings of the DATA step are well documented in the literature.[2] The key concepts are reviewed here, with special emphasis on topics related to MERGE.

Compile and Execute

All DATA steps are compiled in their entirety before being executed. The compilation process defines all variable attributes, including:

1. Name
2. Type
3. Length
4. Label
5. RETAIN vs. reinitialize
6. KEEP/DROP
7. Initial value
8. Format/Informat

For example, this program sets up storage space for the new variable TYPE during the compilation process:

```
DATA PROBLEM;
SET OLD;
IF SEX='M' THEN TYPE='MALE';
ELSE TYPE='FEMALE';
```

A problem arises because TYPE's length is determined by the first DATA step statement capable of defining the length:

```
IF SEX='M' THEN TYPE='MALE';
```

Thus TYPE receives a length of 4 before any data has been read from OLD. The order of the data in OLD is irrelevant to the length of TYPE. The first observation in OLD (in fact, all observations) may contain SEX='F', but TYPE will always have a length of 4.

On the other hand, programs can take advantage of the distinct compilation phase in many ways. For example, the following program might be useful when entering data via SAS/FSP®:

```
DATA YOUR.DATA;
STOP;
SET MY.DATA;
```

In compiling the SET statement, the program reads the header information from MY.DATA, defining all variables. Next, the program executes, and hits the STOP statement. The DATA step is therefore over, and YOUR.DATA contains zero observations, with all variables defined exactly as they exist in MY.DATA. Therefore, data entry via SAS/FSP can begin on YOUR.DATA using the same screens used for MY.DATA. The actual program or programs which created MY.DATA are not needed to create YOUR.DATA.

Here is another example which takes advantage of the distinct compilation phase:[3]

```
DATA _NULL_;
PUT ’TOTAL OBS IS ’  TOTOBS;
STOP;
SET DISK.SASDATA NOBS=TOTOBS;
```

When compiling the SET statement the software can access the header information for DISK.DATASET, including the total number of observations in the data set. The software creates TOTOBS and initializes it with that total number of observations. That value is available to the PUT statement without reading any data values.

Next, consider the execution phase of the DATA step. Some of the important concepts are:

1. Statements which read data (INPUT, SET, MERGE, UPDATE) are executable. They are not merely labels that identify the source of the data. For example, the SET statement means "go out to this data set and read in an observation." Executable statements may appear anywhere in the DATA step, and do not have to be placed right after the DATA statement.

2. The DATA step continually loops through all its statements. The typical way out of this loop (i.e., the typical ending to a DATA step) is for a SET or INPUT statement to fail because there are no more observations left to read.

3. Variables read with a SET, MERGE, or UPDATE statement are retained. That is, their values are not reinitialized to missing just because the program passes through the DATA statement and outputs an observation.

The DATA steps below illustrate these concepts in action.

```
DATA TEMP1;
PUT ’1ST:  ’  _N_=  X=;
INPUT X;
PUT ’2ND:  ’  _N_=  X=;
X=X*10;
PUT ’3rd:  ’  _N_=  X=;
CARDS;
11
22
;

DATA TEMP2;
PUT ’1ST:  ’  _N_=  X=;
SET TEMP1;
PUT ’2ND:  ’  _N_=  X=;
X=X*10;
PUT ’3rd:  ’  _N_=  X=;
```

In comparing the messages generated, notice how SET statement variables are retained.

```
1ST:  _N_=1 X=.
2ND:  _N_=1 X=11
3RD:  _N_=1 X=110
1ST:  _N_=2 X=.
2ND:  _N_=2 X=22
3RD:  _N_=2 X=220
1ST:  _N_=3 X=.

1ST:  _N_=1 X=.
2ND:  _N_=1 X=110
3RD:  _N_=1 X=1100
1ST:  _N_=2 X=1100
2ND:  _N_=2 X=220
3RD:  _N_=2 X=2200
1ST:  _N_=3 X=2200
```

Both DATA steps generate seven messages, not six. Neither step ends until a read statement fails because there are no more incoming data.

Program Data Vector

The PDV is a set of storage locations set up in memory, holding the current value of all variables. Programming statements modify values stored in the PDV, not values stored in SAS data sets. For example, the assignment statement below operates on values stored in the PDV (as opposed to values stored in OLD):

```
DATA TOTALS;
SET OLD;
CUPS=2*PINTS + 4*QUARTS;
```

The program uses the PDV by:

1. Copying observations from OLD into the PDV.

2. Computing CUPS based on the PDV's values for PINTS and QUARTS.

3. Returning to the DATA statement and outputting the current record. This means copying the contents of the PDV (including the computed value for CUPS) to the output data set TOTALS.

For the moment, the key concepts are that the PDV holds the current values for all variables, and that data modification statements affect only values stored in the PDV.

Understanding the PDV clears up otherwise mysterious areas of the SAS software. Consider, for example, the position of these LENGTH statements:

```
DATA NEW;
LENGTH CITY $ 20;
SET OLD;
LENGTH ZIPCODE 4;
```

To change lengths in NEW, the LENGTH statement for the character variable must come before the SET statement but the LENGTH statement for the numeric variable may appear anywhere in the DATA step. The difference is due to the fact that the PDV stores all numerics with a length of 8, regardless of the length in the input or output data sets. DATA step compilation gives CITY a length of 20 in the PDV, and ZIPCODE a length of 8. Only when observations are output, during the subsequent execution phase, does the length of ZIPCODE get truncated to 4. The length of character variables is the same in the PDV and any output data sets.

The role of the PDV clears up confusing combinations of KEEPs, DROPs, and RENAMEs. All KEEPs, DROPs, and RENAMEs on a DATA statement refer to variable names in the PDV. All KEEPs, DROPs, and RENAMEs on a SET, MERGE, or UPDATE statement refer to variable names in the source data set. So when will the following program work?

```
DATA TOTALS (RENAME=(QUARTS=QTS));
SET ALL (DROP=CUPS
         RENAME=(VAR3=VAR4));
CUPS=2*PINTS + 4*QUARTS;
```

These situations would create errors:

- ALL contains QTS or VAR4.
- ALL does not contain VAR3.
- ALL does not contain CUPS.

(The DKRICOND option can control whether the last two items return errors, warnings, or no message at all.) These are not problems:

- CUPS is dropped, and later assigned a value.
- QUARTS is used to compute CUPS, but is later renamed.

All three concepts -- compile and execute, the PDV, and data movement -- are interrelated. Figure 1 shows the use of the PDV in the previous program, and introduces the third key concept: data movement.

Data Movement

As the previous program executes, these steps take place for each observation:

1. Copy all variables except CUPS from ALL to the PDV. When copying VAR3, copy it to the slot labeled VAR4 in the PDV.

2. Calculate and insert (into the PDV) a value for CUPS based on the values of PINTS and QUARTS (in the PDV).

3. Output the observation by copying variables from the PDV to the output data set TOTALS. When copying QUARTS, copy it to the slot labeled QTS in TOTALS.

4. Set CUPS to missing. (Steps 3 and 4 take place at the DATA statement.)

Understanding data movement will let you write more efficient programs as well. Compare these two DATA steps:

```
DATA TOTALS;
SET OLD (DROP=VAR5-VAR20);
CUPS=2*PINTS + 4*QUARTS;

DATA TOTALS (DROP=VAR5-VAR20);
SET OLD;
CUPS=2*PINTS + 4*QUARTS;
```

Figure 1
The PDV and the Flow of Data

```
     OLD               CPU              TOTALS
                 ================
                |                |
                |      PDV       |
 ============      ------------      ============
|    CUPS    |    |    CUPS    |--->|    CUPS    |
|------------|    |------------|    |------------|
|    VAR3    |--->|    VAR4    |--->|    VAR4    |
|------------|    |------------|    |------------|
|    PINTS   |--->|   PINTS    |--->|    PINTS   |
|------------|    |------------|    |------------|
|   QUARTS   |--->|   QUARTS   |--->|   QTS      |
|------------|    |------------|    |------------|
|   Other    |--->|   Other    |--->|   Other    |
|  Variables |    |  Variables |    |  Variables |
 ============     |------------|     ============
                  |  Automatic |
                  |  Variables |
                |  ------------  |
                |                |
                 ================
```

Figure 2
The PDV in a MERGE

```
                       CPU
                 ================
   VACATION     |                |       BOTH
                |      PDV       |
 ============      ------------      ============
|  LOCATION  |--->|  LOCATION  |--->|  LOCATION  |
|------------|    |------------|    |------------|
|    NAME    |--->|            |    |            |
 ============     |    NAME    |--->|    NAME    |
 ============     |            |    |            |
|    NAME    |--->|            |    |            |
|------------|    |------------|    |------------|
|     AGE    |--->|     AGE    |    |            |
 ============     |------------|    |    AGE     |
                  |    DUMMY   |--->|            |
   AGEDATA         ------------      ============
                  |  Automatic |
                  |  Variables |
                |  ------------  |
                |                |
                 ================
```

The PDV for the second program contains VAR5-VAR20, while the PDV for the first program does not. The second program performs extra work, copying those 16 variables from ALL into the PDV for each observation.

At Last, MERGE

The MERGE process employs the concepts above, as well as containing a few of its own characteristics. Consider the following data set and program.

AGEDATA contains one observation per name.

NAME	AGE
ALICE	38
BOB	49
CAROL	55
TED	40

VACATION contains a varying number of observations per NAME:

NAME	LOCATION
ALICE	ARUBA
BOB	BERMUDA
BOB	BIMINI
TED	TAHITI

And the program:

```
DATA BOTH;
LENGTH NAME $ 5 LOCATION $ 8;
PUT 'BEFORE: '  _N_= NAME= AGE= /
    @9 LOCATION= COUNT= X=;
MERGE AGEDATA VACATION;
BY NAME;
COUNT + 1;
X=5;
PUT 'AFTER:  '  _N_= NAME= AGE= /
    @9 LOCATION= COUNT= X=;
```

This program generates the following messages:

```
BEFORE: _N_=1 NAME=  AGE=.
        LOCATION=  COUNT=0 X=.
AFTER:  _N_=1 NAME=ALICE AGE=38
        LOCATION=ARUBA COUNT=1 X=5
BEFORE: _N_=2 NAME=ALICE AGE=38
        LOCATION=ARUBA COUNT=1 X=.
AFTER:  _N_=2 NAME=BOB AGE=49
        LOCATION=BERMUDA COUNT=2 X=5
BEFORE: _N_=3 NAME=BOB AGE=49
        LOCATION=BERMUDA COUNT=2 X=.
AFTER:  _N_=3 NAME=BOB AGE=49
        LOCATION=BIMINI COUNT=3 X=5
BEFORE: _N_=4 NAME=BOB AGE=49
        LOCATION=BIMINI COUNT=3 X=.
AFTER:  _N_=4 NAME=CAROL AGE=55
        LOCATION=  COUNT=4 X=5
BEFORE: _N_=5 NAME=CAROL AGE=55
        LOCATION=  COUNT=4 X=.
AFTER:  _N_=5 NAME=TED AGE=40
        LOCATION=TAHITI COUNT=5 X=5
BEFORE: _N_=6 NAME=TED AGE=40
        LOCATION=TAHITI COUNT=5 X=.
```

Notice the timing of a few key actions:

1. _N_ is incremented each the the DATA step leaves the DATA statement.

2. COUNT is initially 0, and is always retained (never reinitialized).

3. Variables read by the MERGE statement (NAME, AGE, and LOCATION) are retained. They are initially missing, and are reinitialized at the MERGE statement, whenever the program encounters a new value for the BY variable(s).

4. X is reinitialized to missing for each observation, at the DATA statement.

Both the general DATA step processes described above, as well as the MERGE concepts, are important to understanding how MERGE works. However, when programmers begin to apply these concepts, in practice MERGE may produce unwanted results. Most of the time, these results occur in match-merges which either (1) contain a common variable other than the BY variable(s), or (2) are many-to-one MERGEs. The rest of this paper illustrates MERGEs which contain typical problems and shows programming fixes to overcome them.

The first problem area involves common variables other than the BY variable(s). The merged data set contains the last value read from either source data set. In a one-to-one MERGE this means the value from the last data set mentioned in the MERGE statement. But in a many-to-one MERGE the value may come from either data set. Let's once again merge AGEDATA and VACATION:

NAME	AGE
ALICE	70
BOB	70
CAROL	70
TED	70

NAME	LOCATION	AGE
ALICE	ARUBA	38
BOB	BERMUDA	35
CAROL	CANCUN	55
TED	TIMBUKTU	60
TED	TIPPERARY	60
TED	TOLEDO	60

The results depend on the order of the data sets in the MERGE statement.

```
DATA BOTH;
MERGE AGEDATA VACATION
/* or VACATION AGEDATA */;
BY NAME;
```

produces one of these results:

NAME	LOCATION	AGE
ALICE	ARUBA	38
BOB	BERMUDA	35
CAROL	CANCUN	55
TED	TIMBUKTU	60
TED	TIPPERARY	60
TED	TOLEDO	60

ALICE	ARUBA	70
BOB	BERMUDA	70
CAROL	CANCUN	70
TED	TIMBUKTU	70
TED	TIPPERARY	60
TED	TOLEDO	60

The last two AGEs are 60, not 70, because that was the last value read from any of the merged SAS data sets. The AGE of 70 is NOT reread, merely retained in the PDV. When merging in the last two observations, the value of AGE (60) replaces the current value in the PDV.

In many-to-one MERGEs, be careful when modifying variables which come from the "one" data set. Consider one more variation for AGEDATA and VACATION:

NAME	AGE
ALICE	38
BOB	49
CAROL	55
TED	40

NAME	LOCATION
ALICE	ARUBA
ALICE	ARGENTINA
BOB	BAHAMAS
BOB	BERMUDA
BOB	BIMINI
CAROL	CANCUN
TED	TIMBUKTU

The merge BY NAME is straightforward for this many-to-one situation. But suppose the objective were a little more complex, involving some data manipulation to AGE. In particular, suppose BOB’s last vacation was to BERMUDA, and he turned 50 just before he left. The desired result would be:

NAME	AGE	LOCATION
ALICE	38	ARUBA
ALICE	38	ARGENTINA
BOB	49	BAHAMAS
BOB	**50**	BERMUDA
BOB	49	BIMINI
CAROL	55	CANCUN
TED	40	TIMBUKTU

Because AGE is NOT being reread from AGEDATA, but its value is merely being retained in the PDV, the following program will not work:

```
DATA BOTH;
MERGE AGEDATA VACATION;
BY NAME;
IF NAME=’BOB’ AND
LOCATION=’BERMUDA’ THEN AGE=50;
```

The result is that AGE remains 50 for BIMINI, not just for BERMUDA. To get around this problem, it is necessary to create a new variable:

```
DATA BOTH (DROP=AGE
           RENAME=(DUMMY=AGE));
MERGE AGEDATA VACATION;
BY NAME;
IF NAME=’BOB’ AND
LOCATION=’BERMUDA’ THEN DUMMY=50;
ELSE DUMMY=AGE;
```

Figure 2 illustrates the MERGE process for this program.

Finally, consider a combination example where both situations exist: a many-to-one MERGE where both incoming data sets contain a common variable (not the BY variable).

An existing SAS data set MASTER contains many records for each STATE, but does not contain the variable STATE. The objective is to add the STATE variable, based on either of the existing variables COUNTY or ZIPCODE.

Two separate SAS data sets may be able to supply the STATE variable: COUNTIES contains COUNTY and STATE, while ZIPCODES contains three-digit zipcode and STATE. Both data sets are incomplete sources of STATE data, and the information in ZIPCODES is more reliable than the information in COUNTIES. So the plan of action is:

1. Add to MASTER a variable holding the first three characters of ZIPCODE.

2. Sort MASTER and ZIPCODES by shortened ZIP, and MERGE them to get some STATE values added to MASTER.

3. If COUNTIES contains multiple occurrences of a COUNTY, delete all of them. (For example, Suffolk County would appear for both New York and Massachusetts.)

4. MERGE MASTER and COUNTIES by COUNTY. However, if a legitimate STATE value had already been retrieved from ZIPCODES, disregard the STATE from COUNTIES.

In the program for step 1, notice how the LENGTH statement eliminates the need to use a SUBSTR function:

```
DATA MASTER;
SET MASTER;
LENGTH ZIP3 $ 3;
ZIP3=ZIPCODE;
```

Sorting/merging in step 2 is straightforward:

```
PROC SORT DATA=MASTER;
BY ZIP3;

PROC SORT DATA=ZIPCODES;
BY ZIP3;
*CONTAINS ZIP3 + STATE ONLY;

DATA ADDSTATE;
MERGE MASTER (IN=INMAST)
      ZIPCODES;
BY ZIP3;
IF INMAST;
```

As is deleting duplicate counties in step 3:

```
PROC SORT DATA=COUNTIES;
BY COUNTY;
*CONTAINS COUNTY + STATE ONLY;

DATA UNIQUE;
SET COUNTIES;
BY COUNTY;
IF FIRST.COUNTY AND LAST.COUNTY;
```

Step 4 becomes tricky. The idea is to MERGE BY COUNTY and replace only the missing STATE values. Because of the previous MERGE, both data sets now contain STATE. It becomes necessary to RENAME one of the incoming variables:

```
PROC SORT DATA=ADDSTATE;
BY COUNTY;

DATA ADDSTATE (DROP=DUMMY);
MERGE ADDSTATE (IN=INMAST)
      UNIQUE (RENAME=(STATE=DUMMY));
BY COUNTY;
IF INMAST;
IF STATE=' ' THEN STATE=DUMMY;
```

The author welcomes comments and questions about this paper, as well as suggestions for future papers. Feel free to call or write:

Bob Virgile
Robert Virgile Associates, Inc.
3 Rock Street
Woburn, MA 01801
(781) 938-0307
virgile@mediaone.net

Notes:

SAS and SAS/FSP are registered trademarks of SAS Institute Inc., Cary, NC.

[1] Version 6 of SAS software permits storage of compiled DATA step code.

[2] For more details, see Don Henderson's paper, "The SAS Supervisor," published in many SUGI proceedings.

[3] This is not an original program, but has appeared in various forms in past SUGI papers.