Paper 251-30

Let Your Data Power Your DATA Step: Making Effective Use of the SET, MERGE, UPDATE, and MODIFY Statements

Howard Schreier, Howles Informatics, Arlington VA

ABSTRACT

The “plain vanilla” SET statement is ubiquitous in SAS® programming. It feeds the contents of a data set to a DATA step. This usage only scratches the surface of the capabilities of the SET statement and its siblings (MERGE, UPDATE, and MODIFY). Data sets typically are more than collections of individual values; they also incorporate information in the form of structure and organization. The SET, MERGE, UPDATE, and MODIFY statements are versatile tools that can use such information to combine and refine data. This tutorial will explain and illustrate the basic capabilities of each of these statements and will compare their advantages and disadvantages in different programming situations.

INTRODUCTION

We begin with an example of a “plain vanilla” SET statement: First, here is a SAS data set, named IN:

div	store	sales	sq_ft
4	Main	1.56	5.1
4	Downtown	1.12	2.7
7	Airport	0.78	1.1

This simple DATA step uses a SET statement to process the data set:

```
data out;
set in;
ratio = 1000 * sales / sq_ft;
format ratio 8.0;
run;
```

The result is the data set named OUT:

div	store	sales	sq_ft	ratio
4	Main	1.56	5.1	306
4	Downtown	1.12	2.7	415
7	Airport	0.78	1.1	709

It is important to understand that the DATA step language is in fact a procedural language and that the SET statement prescribes actions to be taken. Because the DATA step is equipped with so many default and automatic actions, the code often appears to be declarative. To demonstrate that it is in fact procedural, we can consider the following DATA step, which is essentially equivalent to the one above but which uses explicit code to iterate and terminate and to write out results:

```
data out;
do until (done);
   set in end=done;
   ratio = 1000 * sales / sq_ft;
   output;
   end;
format ratio 8.0;
run;
```

Returning to the first version, we can trace its actions to better understand the central role of the SET statement. First consider the FORMAT statement, which establishes the format for the new variable RATIO. That happens at compile time, so its work is done once execution begins. After execution begins and control reaches the SET statement for the first time, SET gets the first observation from the data set IN. Then control passes to the assignment statement, which computes a value for RATIO, before what is in

effect an implicit OUTPUT statement writes an observation to the data set OUT. Control then returns to the top of the DATA step. The SET statement uses a pointer to track its progress, so it continues with the second observation in IN, which is processed like the first. On the third pass through the DATA step, the SET statement detects that it is operating on the final observation available from data set IN; it records that fact but otherwise does nothing special. Execution continues and a fourth pass is begun. When the SET statement gets control, it knows that it has already processed the last observation available from IN, so it calls for termination of the DATA step. Thus, the SET statement, making use of the automatic looping behavior of the DATA step, has driven this DATA step.

Basics

At this point it will help to be a bit more general, thorough, and precise.

Often, data needed by a DATA step reside in SAS data sets, which in turn are maintained in SAS data libraries. SAS provides library engines to surface the data for use by PROC and DATA steps, one observation at a time.

The program data vector (PDV) is, in effect, a structure maintained in memory while a DATA step is processing. It provides a location for recording the current value of each variable. The PDV is very much the crossroads of the DATA step. A variable's value cannot be used in a formula or in any way written as output unless that value is first stored in the PDV. The PDV (or an adjunct) also keeps track of variable attributes. Attributes are determined when the DATA step is compiled and are locked in before execution begins; this is in contrast to the variables' values, which generally change during execution.

Now we have the context in which we can see the role of the SET statement: it is a tool which copies data surfaced by the library engines to the PDV. It is one of four statements in the DATA step language which fill this role. The others are MERGE, UPDATE, and MODIFY; hereinafter we will use the acronym "SMUM" to refer to the four collectively. There are also function calls which can be coded to extract data from SAS data sets, but the SMUM statements are simpler to use, more powerful, and almost always more appropriate.

Keep in mind that the SMUM statements are executable. Each time a SMUM statement operates during program execution, it accepts and inspects data from the library engine(s), then makes appropriate changes to the PDV.

Several data set options interact with the SMUM statements. Four of these in particular are frequently used:

- WHERE=, which subsets observations,
- DROP= or KEEP=, which subset variables, and
- RENAME=, which renames variables.

These actions are all handled by the engines, so the DATA step only sees the data after they take effect. Another data set option, IN=, is discussed below.

BY processing usually requires that data sets be sorted in the order indicated, or that indices be available to support retrieval in that order.

Consider this DATA step, which processes the data set (IN) used in the first example. It illustrates data set and statement options as well as BY processing:

```
data totals(drop = storesales);
set in(keep = div sales rename = (sales = storesales) ) end=nomore;
by div;
if first.div then sales = 0;
sales + storesales;
if last.div then output;
run;
```

The results (TOTALS):

```
div    sales
 4      2.68
 7      0.78
```

The SET statement can be viewed as managing certain variables in the PDV. In this example, those are
- DIV and STORESALES, which are contributed by the data set being read, as conditioned by data set options;
- FIRST.DIV and LAST.DIV, which report on detection of BY group boundaries; and
- NOMORE, which is created by an option.

Variables which come from data sets via SMUM statements are sometimes said to be "automatically retained". Specifically, this means that they are not initialized to missing at the start of each DATA step iteration. However, they are among those variables managed by the SMUM statement and, as we will see later, this management includes initialization at appropriate times.

It is important to understand that management of variables by SMUM statements is not exclusive. Other user-written code can manipulate these variables. Should one take advantage of this and alter values of variables which are under SMUM management? There is no simple answer. In many situations, it's fairly safe. In other situations, it can lead to error; an example appears later in this paper. In general, the interaction with SMUM statements' automatic actions must be well understood by the programmer.

Other Flavors

We began with the "plain vanilla" SET statement. There are however other flavors offered by the SMUM statements. These fall into three categories:
- combining data from multiple data sets,
- making changes to an existing data set, and
- utilizing random (arbitrary) access to a data set.

Of these, the first is arguably the most important and the most widely useful, so it will be given the most attention.

Before we get into the details, I will use an extended analogy to present the possibilities. Consider the popular Rolodex file. Suppose that yours is in bad shape (full of scribbled notes; containing many cards which are no longer needed; lacking important information, such as up-to-date area codes). You buy a new one so that your assistant can transcribe information and generally fix things up in accordance with your instructions.

Scenario 1. Your assistant processes one card at a time from the old Rolodex, and either discards it or incorporates needed changes in a new card for the new Rolodex. That's like the "plain vanilla" SET statement.

Scenario 2. You actually have two old Rolodexes, one with personal contacts and the other with professional contacts, but you want them combined. The populations are, for the most part, mutually exclusive. Your assistant has to process the cards from both. That's like the SET statement referencing two data sets.

> (A) If you want to keep your personal and professional contacts segregated, you tell your assistant to work through the home Rolodex first, then go on to the office Rolodex, and to set up the new Rolodex with two separate alphabet cycles. That's known as concatenation.
>
> (B) If instead, you decide to intersperse all of the contacts, from both old Rolodexes, in a single alphabet cycle (so that in those cases where a person was in both of the old files, the cards will be adjacent in the new), you tell your assistant to collate the old cards alphabetically before making the new ones. That's called interleaving.

Scenario 3. You have over time inserted a lot of information on friends and family in the office Rolodex, and information on colleagues in the home Rolodex. You want to consolidate everything, so you tell your assistant to pull the old cards and match them up, side by side, and to make just one new card for each person. That's like a typical match-up exercise done with the MERGE statement.

Scenario 4. You have just one old Rolodex, but you also have a stack of business cards, message slips, etc. You want the information from those to be in the new Rolodex, so you tell your assistant to alphabetize the loose items and integrate their content in the new Rolodex. That's like the UPDATE statement.

Scenario 5. You decide that you don't really need a new Rolodex. Instead, you tell your assistant to straighten out the old one. That's like the MODIFY statement.

Scenario 6. Your assistant has the amazing ability to find any particular card in the old Rolodex immediately, with no need to search or to flip back and forth. This is really advantageous if only a few cards need to be processed. That's random access.

COMBINING MULTIPLE DATA SETS

Basically, the SMUM statements move information into place for processing in the DATA step. They also apply structure and organization derived from the data sets. That's true in the "plain vanilla" usage of the SET statement, but it's even more true when you integrate information from two or more data sets. The structure and organization arise from variable attributes, from the sequencing of observations (ie, sort order), from the use of indices, and from the processing rules designed into the statements themselves.

Nevertheless, any SMUM statement "plugs in" to the DATA step in the same way as the simple SET. It is executable, and each time control reaches the statement, it accepts data from the engine(s) and makes the appropriate changes to the PDV. The statement typically (though not always) progresses through the data sets, detects end-of-file conditions, and calls for termination of the DATA step after all data sets have been exhausted.

When a SMUM statement is combining data from two (or more) data sets, there are some issues and concerns which do not arise (or are trivial) in the plain-vanilla situation.

The IN= data set option is often useful when multiple data sets are involved. When it is coded for a particular data set, the tracking variable is initially 0 (zero), but is set to 1 (one) whenever the SMUM statement accepts data from that data set. It will only revert to 0 when the SMUM statement performs what I will call a "reset" on the variables under its management. What do I mean by a reset? Recall that a SMUM statement can be said to manage certain variables. At appropriate times (which can depend on the statement in question, parameters, options, and presence or absence of a BY statement) it will reinitialize some of these in the PDV. Specifically,

- variables whose values come from the SAS data sets are set to missing and
- variables created by the IN= option are set to 0 (zero).

The PDV provides one and only one location for any given variable name. There is no provision for qualification or distinction to accommodate like-named variables from different data sets. In the PDV, values of such variables collide and contend.

Attribute conflicts also become more likely when multiple data sets are involved. They tend to vary in seriousness.

- Type conflicts (numeric vs. character) stop the DATA step at compile time.
- Length conflicts in character variables can corrupt results and generally can have unintended consequences. An example appears below. Length conflicts in numeric variables are less of an issue, because in practice they are less likely to arise and because their effects are quite different in nature; the subject of numeric variable lengths is outside the scope of this paper.
- Other conflicts (as in formats, informats, and variable labels) tend to be less serious but nevertheless deserve programmer attention.

SET Statement

The following two data sets will be used to show how the SET statement can be used to combine data. STATES lists some U.S. states whose names begin with three letters of the alphabet, and a city in each:

```
init    name        city
 f      Florida     Miami
 i      Idaho       Boise
 i      Indiana     Gary
 i      Iowa        Ames
 t      Texas       Dallas
```

COUNTRIES lists some countries whose names begin with the same letters, and the capital of each:

```
init    name        capital
 i      Ireland     Dublin
 i      Italy       Rome
 t      Togo        Lomé
```

To concatenate data sets is to stack them, end to end. That is accomplished by simply naming them in the desired order in a SET statement. For example,

```
data concatenated;
set states countries;
run;
```

yields

```
init    name        city      capital
 f      Florida     Miami
 i      Idaho       Boise
 i      Indiana     Gary
 i      Iowa        Ames
 t      Texas       Dallas
 i      Ireland                Dublin
 i      Italy                  Rome
 t      Togo                   Lomé
```

There are two common variables (INITIAL and NAME), so the values of those from the two data sets are stacked together. The other variables are distinct to one or the other of the original data sets, so missing values arise.

Now suppose that we wanted to collate these observations by initial letter. That is called interleaving, and is accomplished by adding the appropriate BY statement. So,

```
data interleaved;
set states countries;
by init;
run;
```

results in a data set comprising the same observations as the concatenation, but in a different order:

```
init    name        city      capital
 f      Florida     Miami
 i      Idaho       Boise
 i      Indiana     Gary
 i      Iowa        Ames
 i      Ireland                Dublin
 i      Italy                  Rome
 t      Texas       Dallas
 t      Togo                   Lomé
```

The presence of the BY statement governing combination of the data sets imposes a requirement: that SAS be able to retrieve the observations in the order specified. That can be satisfied by sorting the data sets (if necessary), or by having appropriate indices. The requirement also applies to the MERGE and UPDATE statements, but not to the MODIFY statement.

The order in which the observations were sequenced in our interleaving example reflects the rules followed by the SET statement. The first observation accepted is the first one found in the first-named among the

data sets containing observations belonging to the first BY group. If there are additional observations in that data set belonging to that first BY group, they are processed next. When the BY group is exhausted within that data set, the "feed" continues from the next-named data set. This continues until all data sets have been tapped for the first BY group. At that point the process turns to the second BY group and returns to the first-named data set. Note that exhaustion can refer to the situation where a BY group simply has no observations in a particular data set; ie, when a data set has no observations for a particular BY group, it is simply passed over.

Whenever a BY group is begun or the source of observations switches from one data set to another, the PDV is reset (that is, variables which come from the named data sets are initialized to missing and IN= variables are initialized to 0).

This sounds convoluted, but it reflects the behavior of the SET statement in the most general situation (interleaving of multiple data sets). If you understand that process, you will also understand the situations where (1) only one data set is designated on the SET statement and/or (2) there is no BY statement, which constitute simpler special cases.

Here is a variation of the interleaving example, intended to focus on the IN= option:

```
data interleaved;
set states   (keep=init name in=s)
    countries(keep=init name in=c);
by init;
type = 2 * s + c;
run;
```

The result:

```
init    name        type
 f      Florida       2
 i      Idaho         2
 i      Indiana       2
 i      Iowa          2
 i      Ireland       1
 i      Italy         1
 t      Texas         2
 t      Togo          1
```

The sequence of TYPE values reflects the management of the IN= variables by the SET statement. For each of the first two observations (Florida's and Idaho's), S is given the value 1 because the observation begins a BY group and comes from STATES. Consequently TYPE has the computed value 2. For the next two observations (Indiana's and Iowa's), there is no reset, so S and C are not touched by the SET statement and the evaluation of TYPE continues to yield 2. But when Ireland gets its turn, SET notices that a source boundary is being crossed, so it does a reset first. That initializes S to 0 while C is given the value 1; as a consequence, TYPE evaluates to 1.

As noted earlier, whenever a variable is "fed" from two or more data sets, attention should be paid to the attributes as well as to the values. Here is an example in the context of the SET statement; the caveat applies equally to MERGE and UPDATE. The following DATA steps generate data sets having a common variable with different lengths, then concatenate the two:

```
data len5;
length name $ 5;
do name = 'one', 'two', 'three'; output; end;
run;

data len8;
length name $ 8;
do name = 'eleven', 'twelve', 'thirteen'; output; end;
run;
```

```
data both;
set len5 len8;
run;
```

The length of NAME in the PDV in this last DATA step, and thus in the data set BOTH, reflects the length in the first data set designated in the SET statement, LEN5. This results in truncation of some values:

```
name
one
two
three
eleve
twelv
thirt
```

One solution is to code an explicit LENGTH statement. It must appear before the SET statement in order to preempt the length attributes provided by LEN5 and LEN8. So, for example,

```
data both;
length name $ 10;
set len5 len8;
run;
```

yields

```
name
one
two
three
eleven
twelve
thirteen
```

MERGE Statement

The MERGE statement is probably the most intricate of the SMUM statements. We will begin with a couple of examples which are not good application models, but which do illustrate the mechanics.

In its simplest form, the MERGE statement operates in the absence of a BY statement to splice two or more data sets together, side by side. Here is an illustration, using the STATES and COUNTRIES data sets introduced above:

```
data merge_notby;
merge states countries;
run;
```

The resulting data set looks like this:

```
init    name        city     capital
 i      Ireland     Miami    Dublin
 i      Italy       Boise    Rome
 t      Togo        Gary     Lomé
 i      Iowa        Ames
 t      Texas       Dallas
```

It is kind of a mess. The observations are simply paired until the shorter data set is exhausted. Thereafter the output reflects only the longer data set. For the two common variables (INIT and NAME) in the first three observations, there were collisions. SAS had, in each of these 6 cases, two values but only one PDV location in which to store a value. This problem is resolved rather arbitrarily, by favoring the value from the last-named data set. That is why the first three NAME values identify countries. In the last two observations, no observations from COUNTRIES are available, so the values from STATES do not get overlaid and instead survive in the PDV. Because there is no BY statement, each execution of the MERGE

statement begins with a reset of the PDV; that is why, in this example, the last CAPITAL value (Lomé) does not persist.

In practice, this form of the MERGE statement is limited in its usefulness. If a one-to-one correspondence between data sets is assured (or if only one data set contains unmatched observations and these observations are all at the end), the mechanical splice may produce something of value.

Adding a BY statement refines the process and brings us closer to typical usage of the MERGE statement. Consider:

```
data merge_by;
merge states countries;
by init;
run;
```

which results in:

```
init    name         city     capital
 f      Florida    Miami
 i      Ireland    Boise     Dublin
 i      Italy      Gary      Rome
 i      Iowa       Ames      Rome
 t      Togo       Dallas    Lomé
```

The BY statement serves to segregate the processing within each BY group.

In the first BY group (INIT=f), there is no observation from COUNTRIES, so only the values from STATES enter the PDV and wind up in the output data set. In the last BY group (INIT=t), each data set supplies one observation. There is contention for the NAME location in the PDV; “Togo” prevails because COUNTRIES follows STATES in the MERGE statement. Actually, NAME will contain the value “Texas” for an instant after the MERGE statement gets data from STATES, but it is almost immediately overwritten by the value “Togo”, from COUNTRIES.

A many-to-many match, as in the INIT=i BY group in this example, is usually not very useful in applications, but if you understand the mechanics in this situation you should be able to understand the results in other situations (one-to-many, many-to-one, one-to-one, zero-to-some, and some-to-zero) as special cases. Within a BY group, MERGE functions largely as it does without a BY statement, by pairing observations in the sequence in which they present themselves. The difference occurs when one data set is exhausted before the other. Notice that there are two INIT=i cases from COUNTRIES but three from STATES. However, the third observation in this group “inherits” the value CAPITAL=Rome. We will see later that this behavior is useful.

First, let’s try to understand why it happens. When there was no BY statement, the PDV was reset each time the MERGE statement operated. Now, with the introduction of the BY statement, the PDV reset takes place only at the start of each BY group. That’s what permits the persistence of values which we see in the case of CAPITAL=Rome.

In the first pass within this BY group, one observation is accepted from each data set. There is contention for the NAME location in the PDV; “Ireland” prevails because COUNTRIES follows STATES in the MERGE statement. Similarly, on the second pass, another pair of observations is accepted and the collision again obliterates the value of NAME coming from STATES. On the third and final pass within this BY group, there is no new observation available from COUNTRIES, so values are accepted only from STATES. Hence, the value “Iowa” faces no contention and materializes in the PDV. Because there is no reset of the PDV except when a new BY group is begun, the value “Rome”, which had been loaded into the PDV during the previous pass, persists.

Now we turn to some typical MERGE applications. First consider the problem of collating data from two peer data sets, which report on the same set of entities. We do not *a priori* expect the keys from one to be a subset of the keys from the other. To focus on the essence and concentrate on what can be accomplished rather than on what can go wrong, we will avoid both repeating keys and variable collisions.

We start with two data sets, LEFT:

key	left
1	11
2	12
5	15
6	16
7	17
8	18
9	19

and RIGHT:

key	right
0	20
1	21
2	22
7	27
9	29

To integrate all of the information, we apply the MERGE statement in this symmetric fashion.

```
data all;
merge left right;
by key;
run;
```

The result is:

key	left	right
0	.	20
1	11	21
2	12	22
5	15	.
6	16	.
7	17	27
8	18	.
9	19	29

If we wanted just the matching cases, we could have used the IN= data set option thusly:

```
data both;
merge left (in=inleft )
      right(in=inright);
by key;
if inleft and inright;
run;
```

to generate

key	left	right
1	11	21
2	12	22
7	17	27
9	19	29

More intricate variations, with conditionally executed OUTPUT statements and multiple output data sets for the matching and non-matching cases, are also possible.

Next we will look at less symmetric situations. We will see later that some of these are better handled by the UPDATE statement, but we will begin with an example using the MERGE statement in a table lookup operation.

Suppose we are given an extract CUSTOMERS, taken from a customer file:

```
id      state
111      PA
222      NJ
333      PA
444
```

and a table SERVICE_AREA spelling out the state names

```
state_
 code      state
  DE       Delaware
  NJ       New Jersey
  PA       Pennsylvania
```

The task is to add a column to the customer list containing the spelled-out state names. The relationship is many-to-one (a state can be referenced any number of times in the customer file, but appears just once in the lookup table). Moreover, the matching requirements are asymmetric in that all customers are to appear in the output, even if the lookup fails, while it is expected that some lines of the lookup table will be unused, and the consequences of that should be ignored.

The first step is to condition the customer file by means of sorting:

```
proc sort data=customers out=bystate;
by state;
run;
```

Now the MERGE-powered step can run:

```
data named;
merge bystate(in = keeper rename = (state = state_code) )
      service_area;
by state_code;
if keeper;
run;
```

This illustrates the usefulness of data set options in conditioning data to fit requirements. The state abbreviation variables have different names in the two data sets, which prevents them from being used in the BY statement for the MERGE. A RENAME= takes care of that. The renaming is handled by the engine, so the DATA step never even sees the original variable name.

The asymmetry is implemented in the creation of an IN= indicator variable for only the customer file and the subsequent filtering on that variable, which serves to exclude the unused entries in the lookup table (SERVICE_AREA).

The output file should be sorted to restore its original order:

```
proc sort data=named;
by id;
run;
```

The result:

```
        state_
id       code        state
111       PA       Pennsylvania
222       NJ       New Jersey
333       PA       Pennsylvania
444
```

The major causes of trouble in MERGE-driven DATA steps are repeating keys and colliding variable names. Moreover, the two interact in ways which compound the trickiness.

Here is one example. Assume now that we have a data set, DETAIL, which carries spelled-out state names, but with a lot of omissions, errors, and inconsistencies:

id	st_code	state
1	DE	
2	NJ	Jersey
3	PA	PA
4	PA	peensylvania
5	PA	

So, we want to replace those STATE values with ones from a reference data set (RESTATE):

st_code	state	rain
DE	Delaware	4.8
NJ	New Jersey	6.1
PA	Pennsylvania	5.5

Ignore, for now, the variable RAIN. Here is a MERGE-powered DATA step intended to replace the STATE values:

```
data restated;
merge detail restate;
by st_code;
run;
```

Note that RESTATE is named last in the MERGE statement, so that its STATE values will overwrite those from DETAIL. Here is the resulting data set:

id	st_code	state	rain
1	DE	Delaware	4.8
2	NJ	New Jersey	6.1
3	PA	Pennsylvania	5.5
4	PA	peensylvania	5.5
5	PA		5.5

What happened? For the first of the ST_CODE=PA observations, each data set provided a value for STATE and the one from data set RESTATE prevailed, as intended. However, for subsequent ST_CODE=PA observations, only DETAIL provided data, so its faulty STATE values were used.

The solution is to simply exclude the faulty values from the process, by means of a DROP= option:

```
data restated;
merge detail(drop=state) restate;
by st_code;
run;
```

Now there is no contention for the STATE location in the PDV. RESTATE provides its value for STATE at the beginning of each BY group, and it persists through additional observations in that BY group, yielding

id	st_code	state	rain
1	DE	Delaware	4.8
2	NJ	New Jersey	6.1
3	PA	Pennsylvania	5.5
4	PA	Pennsylvania	5.5
5	PA	Pennsylvania	5.5

Now we'll demonstrate another pitfall: manipulating a variable which is also supplied or managed by a SMUM statement. The RAIN variable presumably provides a rainfall measure reported in centimeters, and we wish to convert it to inches. The approach is to code a simple transformation, assigning the result back to the variable RAIN:

```
data restated;
merge detail(drop=state) restate;
by st_code;
rain = rain/2.54;
format rain 5.2;
run;
```

The result:

id	st_code	state	rain
1	DE	Delaware	1.89
2	NJ	New Jersey	2.40
3	PA	Pennsylvania	2.17
4	PA	Pennsylvania	0.85
5	PA	Pennsylvania	0.34

Notice the successively smaller values, unintended, for RAIN in the Pennsylvania group. That happens because RESTATE has just one observation per BY group, so the MERGE statement does not touch the PDV location for RAIN after the first observation in a BY group is processed; rather, it allows the value there to persist. Thus, the transformation is applied repeatedly. One solution is to assign the result of the unit conversion to a new variable.

UPDATE Statement

The UPDATE statement can be seen as a specialized variant of the MERGE statement. UPDATE processes two data sets; no more, no less. Whereas the MERGE statement by default treats data sets as peers, UPDATE has a unique role for each data set. The first-named data set is termed the “master” data set, and the second-named termed the “transaction” data set.

It is probably easier to illustrate the capabilities than to explain them. We will use the CUSTOMERS data set (introduced above) as the master:

id	state
111	PA
222	NJ
333	PA
444	

and the following data set, REV, as the transaction data set:

id	state	limit
111		0
112	DE	5
333	NJ	20
444	MD	.
444		10

We have a new customer (ID=112), a change (the STATE value for ID=333), a replacement for a missing value (the STATE value for ID=444), and a new variable (LIMIT). We can apply these revisions by driving a DATA step with the UPDATE statement:

```
data customers_rev;
update customers rev;
by id;
run;
```

The result (CUSTOMERS_REV):

```
id      state    limit
111      PA         0
112      DE         5
222      NJ         .
333      NJ        20
444      MD        10
```

Notice that SAS was smart about a couple of things. First, the missing values for STATE in REV did not replace the corresponding values in CUSTOMERS; only non-missing values from REV were used. Also, the two observations for ID=444 were "rolled up" automatically (although two iterations of the DATA step were involved).

With the UPDATE statement, a BY statement is mandatory. At the start of each BY group, the PDV is reset and, if there is an observation from the master data set, it is used to load the PDV. Then, during that same execution of UPDATE, if there are observations from the transaction data set, the first of these is used to load the PDV, possibly overlaying values from the master. Subsequent executions of the UPDATE statement advance through any additional transaction-side observations within the current BY group. Thus, the effects of multiple transaction observations are cumulative (in the sense that later supercedes earlier; explicit code is required to implement additive accumulation). If there are additional, extraneous master-side observations within the same BY group (a condition which basically violates the UPDATE statement's process model and which therefore triggers a warning in the log), a reset is done before each; since the transaction-side observations have already been used up by the first master-side observations, there is in effect no updating of these additional observations.

Notice that variable collisions, which are often troublesome with MERGE, are actually expected when UPDATE is used.

When there are multiple transaction-side observations for a BY group, the DATA step will iterate for each one, but will in the absence of an OUTPUT statement write to the output data set only after processing the last. However, if one codes an explicit OUTPUT statement, the intermediate observations will also be written out. So, for example, this DATA step:

```
data customers_rev;
update customers rev;
by id;
if state='MD' then output;
run;
```

Yields this result:

```
id      state    limit
444      MD         .
444      MD        10
```

This is really not a consequence of the UPDATE statement's behavior. It has more to do with hidden coordination between UPDATE and the SAS supervisor. One solution would be to simply delete "then output", converting the conditional explicit OUTPUT statement to a subsetting IF statement.

The four SMUM statements have distinct behaviors and purposes, and it is usually easy to decide which should be used in a particular situation. This tends to be less true when the choice is between MERGE and UPDATE, and as a consequence UPDATE is probably underutilized. Keep in mind the three distinctive characteristics of UPDATE:

- one-to-many master/transaction relationship
- "smart" treatment of missing values (as a consequence of UPDATEMODE=MISSINGCHECK, in effect by default), and
- automatic rollup, with one observation generated per BY group.

If all of these are appropriate to the task at hand, you probably should use UPDATE rather than MERGE.

CHANGING A DATA SET IN PLACE

The MODIFY statement is a relatively recent addition to the DATA step language. Its distinctive feature is that it applies changes (insertion, deletion, and revision of observations) to existing data sets, in place. In other respects, MODIFY resembles SET (if one data set is referenced) or UPDATE (if two data sets are referenced). To a degree, MODIFY gives DBMS-like functionality to the DATA step language.

Looking at a DATA step, there are three possible relationships between an input data set and an output data set.

- They are separate and distinct (by virtue of having different names, or residing in different libraries, or both).
- They are separate, but have the same name and reside in the same library, and so are not distinct. In this case the output data set is built using an automatically generated temporary name; after the data set is complete, the input data set is deleted and the output data set is renamed. This housekeeping is transparent to the user.
- They are one and the same. Only the MODIFY statement implements this relationship.

In its simplest form, MODIFY designates only one data set (which must also be designated on the DATA statement) and thus does no combining. This usage is in many respects much like the “plain vanilla” SET statement. Here is an example illustrating the essential difference. Begin with data set BIG:

```
i
1
2
3
4
5
6
7
8
```

Then run this SET-powered DATA step, which transforms some of the observations, then stops:

```
data big;
set big;
if i<6 then i = i + 0.1;
else stop;
run;
```

The result:

```
i
1.1
2.1
3.1
4.1
5.1
```

Because the step halted during the sixth iteration, there are only five observations in the new data set. Now we switch from SET to MODIFY (and lower the threshold to 4, so that we will be able to see the effect):

```
data big;
modify big;
if i<4 then i = i + 0.1;
else stop;
run;
```

The result:

```
i
1.2
2.2
3.2
4.1
5.1
```

This DATA step worked on BIG in place, so even though it stopped processing during the fourth iteration, the unaltered observations remain.

MODIFY also operates in a master/transaction mode, much like that provided by UPDATE. In fact, we can adapt the example used to demonstrate UPDATE. The master data set is CUSTOMERS:

```
id      state
111      PA
222      NJ
333      PA
444
```

and the transaction data set is REV

```
id      state    limit
111                 0
112      DE         5
333      NJ        20
444      MD         .
444                10
```

Here is the MODIFY-powered DATA step:

```
data customers;
modify customers(in=id_exists) rev;
by id;
if id_exists then replace;
else do;
   output;
   _error_ = 0;
   end;
run;
```

The MODIFY statement (like SET, MERGE, and UPDATE) works with the OUTPUT statement. In addition, it has two other actions available: REPLACE and REMOVE. In this example, it is necessary to detect whether an ID value is present in the target data set, and to use REPLACE or OUTPUT accordingly. Since we expect to process ID values which are not in the master, the automatic variable _ERROR_ is reset to 0 to suppress unwanted log messages.

The result:

```
id      state
111      PA
222      NJ
333      NJ
444      MD
112      DE
```

The fact that SAS is making changes to an existing data set introduces a number of constraints. New observations (such as ID=112) go at the end. Variables cannot be added, so although LIMIT was in the PDV and thus available during DATA step processing, it is not in the output data set.

MODIFY does not require either data set (master or transaction) to be sorted or indexed, though an index can improve performance. SAS examines the transaction-side observations in order, one each time that control passes to the MODIFY statement. When a BY boundary is crossed, the PDV is reset. Using the BY variable values from the transaction-side observation, a search is made for the first matching observation in

the master data set. The _IORC_ automatic variable and the IN= indicator variable (if any) are set to indicate the results of the search. If the observation is found, it is used to load the PDV. Then the transaction-side observation is used to load the PDV, possibly overlaying master-side data. At that point, MODIFY has finished its work on the PDV; the subsequent execution of OUTPUT, REMOVE, or REPLACE statements determines what is done to the master data set. The default, in the absence of any such explicit statement, is to replace the observation when control reaches the end of the DATA step.

With MODIFY, you are "working without a net". If some system failure occurs while the step is processing, the data set could be corrupted. If your program logic is flawed, data could be lost or corrupted. So you probably want to use only well-tested code with MODIFY, and have sound data backup arrangements.

Although MODIFY does not require that data sets be sorted or indexed to support BY processing, there are performance considerations. Consider the following test:

```
data long;
do id = 1 to 1e5;
   output;
   end;
run;

data short;
do id = 1e5-99 to 1e5;
   output;
   end;
run;
```

LONG has 10,000 observations and short has 100, which match the last 100 in LONG. The following DATA step does not accomplish anything, but it does go through the motions of loading into the PDV each matched observation in LONG and then replacing it, unchanged, in the stored data set.

```
data long;
modify long short;
by id;
run;
```

The step took 18 seconds to complete, even though both data sets are in ID order. For each observation in SHORT, there is a sequential search for the matching observation in LONG. It seems that each search starts at the beginning of LONG. Now index LONG and rerun the MODIFY-powered step:

```
proc data sets;
modify long;
index create id;
quit;

data long;
modify long short;
by id;
run;
```

The execution time is reduced to a fraction of a second. The implication is that data sets intended to be used as master data sets with the MODIFY statement ought to be indexed.

RANDOM ACCESS

The SET and MODIFY statements can be coded to ask the engine to retrieve any arbitrary observation, either in terms of its sequential number (using the POINT= option) or in terms of a key invoking an index (using the KEY= option). I use the adjective "random" (rather than "direct") to describe this mode of usage in order to distinguish it from the way in which BY processing uses an index to perform retrieval which is

physically direct but logically sequential. When the POINT= or KEY= option is used, the access is neither physically nor logically sequential.

Here is an example which will demonstrate these two uses of an index. Begin with a data set having its key values in a scrambled order:

```
data list(index=(id) );
do id = 0,8,3,2,1,9,7,4,6,5;
   square = id**2;
   output;
   end;
run;
```

First, we'll use a BY statement to process the data set in ID order, thanks to the existence of the index:

```
data inorder;
set list;
by id;
run;
```

The result:

id	square
0	0
1	1
2	4
3	9
4	16
5	25
6	36
7	49
8	64
9	81

Now we'll use the KEY= option to process observations in arbitrary order:

```
data arbitrary;
do id = 8,5,2;
   set list key=id / unique;
   output;
   if _n_=2 and id=5 then stop;
   end;
run;
```

The output:

id	square
8	64
5	25
2	4
8	64
5	25

It's important to understand that these random-access forms of SET and MODIFY do not drive the DATA step; they neither provide a processing sequence nor terminate execution.

MORE

This paper was intended to focus on the essential nature and purpose of the SET, MERGE, UPDATE, and MODIFY statements. It is not a reference and does not provide a comprehensive explanation of these statements or their interaction with other features of the DATA step language. The reader is encouraged to consult the documentation and the user-written literature (conference proceedings and newsgroup/list archives).

Examples were kept simple. There is always a danger that simplifications will be viewed as implicit declarations of limitation. Don't assume that you can't do something, just because you do not see a statement or example telling you that you can. Some relatively obvious examples: you can code multiple variables on a BY statement, and create corresponding composite indices; data set options can be used on output data sets as well as input data sets. A less obvious example: a DATA step can include more than one SMUM statement (eg, two SET statements, or a SET statement and a MERGE statement). In fact, there are well known techniques employing each of these arrangements.

The paper advised avoiding certain practices (eg, manipulating variables which are managed by the SMUM statements). However, a "never say never" perspective is appropriate. It's just important to understand consequences so that unintended results can be avoided.

The scope of the paper precluded discussion of other SAS tools (most notably, PROC SQL) which are applicable to data manipulation tasks like those performed by the SMUM statements.

REFERENCES

Kuligowski, Andrew (2004), "How to Incorporate Old SAS Data into a New DATA Step," *Proceedings of the Twenty-Ninth Annual SAS Users Group International Conference*
<http://www2.sas.com/proceedings/sugi29/254-29.pdf>

Moorman, Denise and Warner, Deanna (July 1999), "Updating Data Using the MODIFY Statement and the KEY= Option," *Observations: The Technical Journal for SAS Software Users*
<http://support.sas.com/documentation/periodicals/obs/obswww19/index.html>

SAS Institute Inc. (1993), "Performance of MODIFY vs. UPDATE," SAS Notes (V6-SYS.DATA-7118)
<http://support.sas.com/techsup/unotes/V6/7/7118.html>

SAS Institute Inc. (2004), *SAS® 9.1.3 Language Reference: Concepts*.
<http://support.sas.com/onlinedoc/913/docMainpage.jsp>

SAS Institute Inc. (2004), *SAS® 9.1.3 Language Reference: Dictionary, Volumes 1, 2, and 3.*
http://support.sas.com/onlinedoc/913/docMainpage.jsp

SAS Institute Inc. "DATA Step Programming Using the MODIFY Statement," TS Docs (TS-250)
<http://support.sas.com/techsup/technote/ts250.html>

Virgile, Bob (1999), "How MERGE Really Works," *Proceedings of the Twelfth Annual Northeast SAS Users Group Conference*
<http://www.nesug.org/html/Proceedings/nesug99/ad/ad155.pdf>

Whitlock, Ian (1997), "A SAS Programmer's View of the SAS Supervisor," *Proceedings of the Twenty-Second Annual SAS Users Group International Conference*
<http://www2.sas.com/proceedings/sugi22/ADVTUTOR/PAPER34.PDF>

ACKNOWLEDGMENTS

Ian Whitlock and Marianne Whitlock provided helpful comments on preliminary parts of this paper.

CONTACT INFORMATION

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

Howard Schreier
Howles Informatics
Arlington VA

703-979-2720

hs AT howles DOT com
http://www.howles.com/saspapers/