Paper 167-2008

SET, MERGE and beyond

Erik W. Tilanus, consultant, Driebergen, the Netherlands

ABSTRACT

You know the SET statement and the MERGE statement. Is there more to it?
Well, yeah there are a few more tricks to know.
We will discuss random access to SAS data sets, based on observation numbers or indices and we will create a “look ahead” to see what is in the next observation (more or less the opposite of the LAG function). We will explain how to prevent the well-known note in the SAS Log: “MERGE statement has more than one data set with repeats of BY values”.
Then it is time to move on to the UPDATE and MODIFY statements. We will discuss their features and the differences between these statements and SET and MERGE.

INTRODUCTION

So you know how to work with the SET and MERGE statement. Well let us see about that. Before continuing, try the following exercise:
We have two SAS data sets, A and B:

Data set: A

ID	X	Y
01	12	11
02	15	.

Data set B

ID	X	Z
01	.	4
03	17	6
03	18	.

Try to reason what data set C looks like after:

a.
```
DATA C;
   SET A B;
   RUN;
```

b.
```
DATA C;
   SET A;
   SET B;
   RUN;
```

c.
```
DATA C;
   SET A B;
   BY ID;
   RUN;
```

d.
```
DATA C;
   MERGE A B;
   RUN;
```

e.
```
DATA C;
   MERGE A B;
   BY ID;
   RUN;
```

f.
```
DATA C;
   UPDATE A B;
   BY ID;
   RUN;
```

g.
```
DATA C;
   SET B;
   IF X=. THEN SET A;
   RUN;
```

h.
```
DATA C;
   SET B;
   IF X NE . ;
   RUN;
```

i.
```
DATA C;
   SET B;
   IF X EQ . THEN DELETE;
   RUN;
```

j.
```
DATA C;
   MERGE A(IN=IN_A) B(IN=IN_B);
   BY ID;
   IF IN_A AND IN_B;
   RUN;
```

k.
```
DATA C;
   DO UNTIL (ID EQ LAG(ID));
   SET B;
   END;
   RUN;
```

You can look in the appendix at the end of this paper to see whether you were right. Got all of them correctly? Wow! Well done!

Now let us move on to more applications, tips and tricks with SET, MERGE, UPDATE and MODIFY.

THE SET STATEMENT

The SET statement is the main tool to read observations from a SAS data set. The basics are so obvious you might not even consider them. Just SET and a data set name and there you go. But let us have a little look under the hood.

THE COMPILATION PHASE

During DATA step compilation, SAS opens the data set(s) that have been declared in SET statement(s). SAS determines what variables there are and the number of observations from the data set description. The variables are subsequently declared in the Program Data Vector (taking into account DROP, KEEP and RENAME data set options), the buffer space in the computer memory where all variables are maintained. All variables will also have the RETAIN switch turned on, i.e. variables read from a SAS data set are automatically retained. Normally you don't notice this, but there are exceptions. We will discuss an example of that later. The number of observations is recorded in an internal variable and if the NOBS=variable option is specified it will also be recorded in the declared variable. Finally a pointer is initialized which during the execution phase keeps track of how far the reading of the data set has progressed.
The variable declared in the NOBS option will not be included in the output data set, but can be used in the DATA step as a normal numeric variable.

THE EXECUTION PHASE

In principle SAS reads sequentially through the data set declared in a SET statement; each subsequent execution of the SET statement (normally the next iteration of the DATA step) reads the next observation until an End-of-File (EOF) indicator is reached. Since each SET statement has its own pointer, a data set can be read by several SET statements at the same time, with each SET statement reading at a different point. With the following construction the opposite of the LAG function can be achieved, namely a 'look ahead':

```
...
SET MyDataset(FIRSTOBS=2 RENAME=(MyVariable=NextValue));
SET MyDataset;
StepValue = NextValue - MyVariable;
...
```

Because both SET statements are always executed, the first steps ahead one observation each time. Watch out at the end: the first SET statement shall also first switch on an EOF indicator, which means that the last observation in the data set will not be read in the second SET statement, unless you take special measures (e.g. execute the second SET statement twice if the first signals EOF through the END= option, next paragraph). With the POINT option in the first SET, you could also prevent the EOF indication.

MULTIPLE DATA SETS IN A SET STATEMENT

When you specify more than one data set in a SET statement, the mentioned data sets will be read in the sequence in which they are mentioned. The PDV will contain all variables that exist in any of the mentioned data sets (unless controlled via DROP or KEEP options). If a particular variable is not present in some of the data sets, its value will be "missing" while reading observations from that data set. This process is called concatenation of data sets.
A special situation occurs when you combine the SET statement with a BY statement. In the BY statement you specify one or more common variables in the data sets. Now the data sets are not concatenated, but interleaved. The observations are read from the specified data sets in the order of the value of the variables in the BY statement.
We will demonstrate the two situations based on the data sets in Figure 1. It shows the stock listing of two stores of a Do-It-Yourself retail chain. If you combine the two data sets with a straightforward DATA step like:

```
DATA TotalStockList;
SET Store1 Store2;
RUN;
```

TotalStockList will first contain all observations from Store1, followed by the observations from Store2.
If we add a BY statement, like in:

```
DATA TotalStockList;
SET Store1 Store2;
BY Article_number;
RUN;
```

the total list will be in article number order, taking observations from both data sets in the sequence of the article number. Note that the original data sets are in article number sequence. If that was not the case, you should have sorted the data sets before you can combine them this way.

Figure 2 shows the result of the above mentioned DATA steps. In the left column the data sets are simply

concatenated, first store 1001 then store 1002. The right column shows the total list ordered by article number, placing the stock levels of the same article number of the two stores next to each other.

Figure 1: Stock listings of two branches of a DIY-shop

Obs	Store Number	product_ category	article_ number	stock_ level
1	1001	timber	10020	4
2	1001	timber	10030	9
3	1001	timber	10050	12
4	1001	sheet	11030	8
5	1001	sheet	11070	16
6	1001	paint	20060	17
7	1001	paint	20100	8
8	1001	paint	21090	9
9	1001	paint	22001	3
10	1001	tools	40090	4
11	1001	tools	45030	9
12	1001	hardware	50010	15
13	1001	hardware	50030	17

Obs	Store Number	Product_ Category	Article_ Number	Stock_ Level
1	1002	timber	10020	11
2	1002	timber	10030	21
3	1002	sheet	11030	18
4	1002	sheet	11070	9
5	1002	paint	20060	2
6	1002	paint	20100	16
7	1002	paint	21090	12
8	1002	paint	22001	3
9	1002	tools	40090	8
10	1002	tools	45030	12
11	1002	tools	45130	5
12	1002	hardware	50010	9
13	1002	hardware	50030	13

END= OPTION

The END= option signals the end of the input processing. It is similar to the END option for the INFILE statement and works with the SET, MERGE and UPDATE statements. The format is:

```
END = EndFlagVariable
```

The EndFlagVariable has the value 1 for the final observation and 0 in all other cases. If multiple data sets are called, then the END-variable will be 1 after the reading of the final observation of the final data set. The END= option is not a data set option, but a specific SET, MERGE and UPDATE option. The option therefore does not go in parentheses, but as normal at the end of the statement, e.g.:

```
SET MyDataSet1 MyDataSet2 END=EndFlag;
```

IN= OPTION

Like the END option you specify a variable in the IN option that will have a value 1 if SAS is reading from the corresponding data set and 0 otherwise. IN= is a data set option. So you specify it between parentheses after the data set name, like in:

```
SET MyDataSet1(IN=Set1) MyDataSet2(IN=Set2) END=EndFlag;
```

While reading MyDataSet1 the variable Set1 will have the value 1 and Set2 the value 0 and while reading MyDataSet2 it is the opposite. The IN= option is also useful with the MERGE statement. We will discuss an example in that section.

FIRST. AND LAST. VARIABLES

When using the BY statement in conjunction with a SET or MERGE statement, the SAS System creates two special variables which mark the first and last observation of a BY group. These variables are named FIRST.ByVariable and LAST.ByVariable, where ByVariable is the name of the BY variable that determines the BY-group separations you want to catch.

The FIRST.variable has a value 1 if the related BY variable (or a BY variable higher in the BY hierarchy, i.e. earlier in the BY statement) has changed value, and otherwise is 0. The LAST.variable just has a value 1 for the last observation of a BY group and otherwise 0.

The following statements can be used to test whether a BY group consists of only one observation:

```
SET ...;
BY ByVariable;
IF FIRST.ByVariable AND LAST.ByVariable THEN ...;
```

In the paragraph on 'Repetition of BY values', later in this paper another application of FIRST. and LAST is demonstrated.

Figure 2: Effect of SET without and with BY statement

SET Store1 Store2, without BY statement

Obs	Store Number	Product_ Category	article_ number	stock_ level
1	1001	timber	10020	4
2	1001	timber	10030	9
3	1001	timber	10050	12
4	1001	sheet	11030	8
5	1001	sheet	11070	16
6	1001	paint	20060	17
7	1001	paint	20100	8
8	1001	paint	21090	9
9	1001	paint	22001	3
10	1001	tools	40090	4
11	1001	tools	45030	9
12	1001	hardware	50010	15
13	1001	hardware	50030	17
14	1002	timber	10020	11
15	1002	timber	10030	21
16	1002	sheet	11030	18
17	1002	sheet	11070	9
18	1002	paint	20060	2
19	1002	paint	20100	16
20	1002	paint	21090	12
21	1002	paint	22001	3
22	1002	tools	40090	8
23	1002	tools	45030	12
24	1002	tools	45130	5
25	1002	hardware	50010	9
26	1002	hardware	50030	13

SET Store1 Store2, with BY statement

Obs	Store Number	Product_ Category	Article_ Number	Stock_ Level
1	1001	timber	10020	4
2	1002	timber	10020	11
3	1001	timber	10030	9
4	1002	timber	10030	21
5	1001	timber	10050	12
6	1001	sheet	11030	8
7	1002	sheet	11030	18
8	1001	sheet	11070	16
9	1002	sheet	11070	9
10	1001	paint	20060	17
11	1002	paint	20060	2
12	1001	paint	20100	8
13	1002	paint	20100	16
14	1001	paint	21090	9
15	1002	paint	21090	12
16	1001	paint	22001	3
17	1002	paint	22001	3
18	1001	tools	40090	4
19	1002	tools	40090	8
20	1001	tools	45030	9
21	1002	tools	45030	12
22	1002	tools	45130	5
23	1001	hardware	50010	15
24	1002	hardware	50010	9
25	1001	hardware	50030	17
26	1002	hardware	50030	13

NOBS= OPTION

As mentioned with the compilation actions of the SET statement, you can include the NOBS option:

```
SET ... NOBS=ObsCount;
```

SAS will set the variable you specify in the NOBS option to the number of observations in the data set. This option is very useful if you are working with random access to the data: it tells you the highest observation number you can access.

RANDOM ACCESS: POINT AND KEY OPTIONS

The POINT option

Up to now the data set has always been read sequentially by the SET statement. In many cases a SAS data set can also re read in random order. This cannot be done with compressed data sets or data sets on tape, but with SAS data

sets on disk it is possible. The first possibility is using the POINT option:

```
SET DataSet POINT=PointerVariable;
```

in which PointerVariable is a numeric variable. The value of the pointer variable is the observation number that will be read. This makes it possible to read in random order, by changing the pointer variable at random. The reading of the last observation no longer automatically means the end of the operation. The consequence is that the automatic flagging of end-of-file does not work anymore. You have to stop the DATA step manually with a STOP statement. Refer to the section on taking random samples from a data set for an example of using the point option.

The KEY option

If an index to a SAS file has been constructed (by means of the INDEX=data set option or with an INDEX CREATE statement in PROC DATASETS), the index variable can also be used as basis for random access to the data set, with the KEY option:

```
SET dataset KEY=index;
```

in which index is the name of the index variable or the 'composite key'. The index variable(s) is (are) provided with the key values to be found before the SET statement. The first execution of a SET statement with a fixed key value yields the first observation meeting the criterion. Each subsequent execution of the SET statement with the same key value automatically gives the next observation. When all observations of the key value have been read, SAS will return an error code in the _IORC_ variable (see ahead) and set the _ERROR_ variable. The next time the SAS starts over at the beginning of the group. If another key value is requested and after that you revert back to the former, then the system restarts at the beginning of the group.
You can change this behaviour by adding the option /UNIQUE to the SET statement. In that case each execution of the SET statement will reset the pointer and come up with the first matching observation. This method of accessing data sets is very suitable for consulting tables; refer ahead to the section 'Tables look-up'.

It is important to test the automatic variable _IORC_ when using the KEY option. This has the value 0 if the execution of the SET statement has been successful. At the End-of-File indicator the value is -1. Other values of _IORC_ indicate an unsuccessful execution of the SET statement and here the value of the variables is unpredictable. We will go further into this in our discussion of the MODIFY statement later in this paper.
In case the repetition of key values is legitimate, you may want to process the observations with the common key in a loop like:

```
DO UNTIL (_IORC_  NE 0);
   SET ... KEY=keyvalue;
   IF _IORC_ = 0 THEN DO;
      * further processing;
   END;
END;
_ERROR_ = 0;
```

The _ERROR_ variable is reset to zero to prevent a "false alarm" at the end of the step that something went wrong. If you have to deal with a situation where you have two (or more) nested DO loops of this kind, you should also reset _IORC_ back to zero after the inner loop. Otherwise the non-zero _IORC_ from the inner loop will signal the end of the outer loop as well!

THE MERGE STATEMENT

While the SET statement reads observations from one data set at the time, the MERGE statement reads from two or more data sets simultaneously. The PDV contains all variables from all data sets: you are combining the observations from the data sets in the MERGE statement into one observation.

Although BY is not mandatory, you will hardly see a MERGE without a BY statement. Without the BY statement MERGE will execute a "one to one" merge: it combines simply the first observation of all mentioned data sets, then the second observation and so on.
With a BY statement the observations will be matched based on identical values of the BY variables, so the BY variables function as the linking pin between the data sets.

If there are more common variables than those in the BY statement you may run into trouble. Since SAS can only have one value for each variable, there is the convention that only the value in the last mentioned data set will be saved. If you want to maintain the values of the matching variables of each data set, you will have to rename the

variables.
We will demonstrate the MERGE statement also with the stock listings of Figure 1. (Program 1) We use the article number as the linking pin between the data sets. Since the data sets have an identical structure the other variables are also common. For the variable Product_Category this is not important: if the article numbers match, the product category will match as well. We will rename the variables StoreNumber and Stock_Level in the second data set, to be able to maintain them both. At the same time we will calculate the total stock per article number.
Figure 3 shows a PROC PRINT output of the above step. Note that if a certain article is not available in one of the stores, the values, the corresponding variables show missing values.

Program 1: A MERGE of two data sets with common variables

```
DATA TotalStockList;
MERGE Store1 Store2(RENAME=(StoreNumber=StoreNumber2
                           Stock_Level=Stock_level2));
BY Article_number;
Total_Stock = SUM(Stock_Level,Stock_Level2);
RUN;
```

Figure 3: MERGE of two data sets.

Merge of Store1 and Store2 data sets 1
11:28 Saturday, October 7, 2006

Obs	Store Number	Product_ Category	Article_ Number	Stock_ Level	Store Number2	Stock_ level2	Total_ Stock
1	1001	timber	10020	4	1002	11	15
2	1001	timber	10030	9	1002	21	30
3	1001	timber	10050	12	.	.	12
4	1001	sheet	11030	8	1002	18	26
5	1001	sheet	11070	16	1002	9	25
6	1001	paint	20060	17	1002	2	19
7	1001	paint	20100	8	1002	16	24
8	1001	paint	21090	9	1002	12	21
9	1001	paint	22001	3	1002	3	6
10	1001	tools	40090	4	1002	8	12
11	1001	tools	45030	9	1002	12	21
12	.	tools	45130	.	1002	5	5
13	1001	hardware	50010	15	1002	9	24
14	1001	hardware	50030	17	1002	13	30

CHECKING MATCHING IF OBSERVATIONS, USING THE IN= OPTION

Assume you are creating a periodic product catalog. You have compiled the new version and now want to check the differences with the old version. Let us write a program that creates a data set with all the contents of the new catalog and per product (=per observation) a note whether the product is "CONTINUED", "DISCONTINUED" or "NEW". Program 2 shows how this can be accomplished. We simply merge the old and new version of the catalog by ProductCode. If the particular product code is present in the old catalog, variable "old" is set to 1 and if the article is present in the new catalog, "new" is set to 1. So if the are both 1, the article is continued. Else we test whether it was present in the old version. In that case it is discontinued. All other cases it must be new.

Program 2: Use of IN= option to determine the source of the information

```
DATA Comparison;
LENGTH Comment $12;
MERGE OldCatalog(IN=old) NewCatalog(IN=new);
BY ProductCode;
IF old AND new THEN comment = 'CONTINUED';
ELSE IF old THEN comment = 'DISCONTINUED';
ELSE comment = 'NEW';
RUN;
```

"REPEAT OF BY-VALUES" WARNING AND SOLUTIONS

A common note in the SAS Log when using the MERGE statement is "NOTE: MERGE statement has more than one data set with repeats of BY values." Although it is a note, it should be considered an error in most cases.
When there is a repetition of BY values in more than one data set, SAS switches to 'one to one merging', i.e. it steps through all data sets one observation at the time and matches them. When the last observation of that particular BY

value is reached for a data set, it will be used further until it has been matched to the remainder of the observations for that BY group in the other data set(s).
The following example shows how errors get introduced and how they can be avoided.
Assume a data set Jobs containing a number of tasks to be executed. Each task consists of a number of subtasks whose run times are known. The subtasks and their run times are contained in data set Norm_Times (Figure 4).
Requested is to develop a program that calculates the total time for the tasks in Jobs.

Figure 4: Data sets to demonstrate "Repeat of BY values"

Data set Jobs

Obs	Task
1	A
2	A
3	A
4	B

Data set Norm_Times

Obs	Task	Subtask	Duration
1	A	A1	10
2	A	A2	5
3	B	B1	20
4	B	B2	15
5	B	B3	10

A simple MERGE would look like this:

```
DATA WrongResult;
   MERGE Jobs Norm_Times;
   BY Task;
   TotalWork + Duration;
RUN;
PROC PRINT;
RUN;
```

This data step results in the output as printed in Figure 5

Figure 5: Effect of a simple MERGE with repeat of BY variables.

Obs	Task	Subtask	Duration	Total Work
1	A	A1	10	10
2	A	A2	5	15
3	A	A2	5	20
4	B	B1	20	40
5	B	B2	15	55
6	B	B3	10	65

Since there is a repeat of BY values for task A, the first observation of data set Jobs is merged with the first observation of Norm_Times and the second observation of Jobs to the second observation of Norm_Times. Then the end of the BY group in Norm_Times is reached, so the third observation in Jobs is also matched to the second observation in Norm_Times. As a result the second and third task A are calculated incorrectly.

There are several options to avoid these problems. One method would be to calculate first the total time per task, and store that in an intermediate data set. Then use this data set to merge to Jobs. Since the intermediate data set contains only one observation per task, this will work without a problem. Program 3 shows the program. The first DATA step reads the Norm_times data set (Figure 4) sums the time needed for each task and it will write an observation to TaskTime when all subtasks have been read. The second DATA step merges Jobs and TaskTime and since TaskTime now has only one observation per task there is no warning about the repeat of by values. Figure 6 shows the result: this time the correct figures.

Program 3: Avoiding "MERGE statement has more than one data set with repeats of BY values."

```
DATA TaskTime (KEEP = Task TaskTime);
   SET Norm_times;
   BY Task;
   IF FIRST.Task THEN TaskTime = 0;
   Tasktime + Duration;
   IF LAST.Task THEN OUTPUT;
RUN;
DATA CorrectResult;
   MERGE Jobs TaskTime;
   BY Task;
   TotalWork + TaskTime;
RUN;
```

Figure 6: Correct output of Job/Task calculation

Obs	Task	Task Time	Total Work
1	A	15	15
2	A	15	30
3	A	15	45
4	B	45	90

MXN MERGE

Assume that you have a repeat of BY values in more than one data set and you want to combine all observations of one data set with all corresponding observations of a second data set. To accomplish this, one of the options is to make an index data set on one data set, which contains per BY value a pointer to the first and last observation of each BY group. Then you can merge that data set with the other data set and read the observations behind the index with a SET with POINT option.

Program 4 applies this technique on the Jobs and Norm_Times data sets from the previous paragraph.

Figure 7 shows the results.

First we create the index based on the Norm_Times data set. It will contain two observations, one for task A and one for task B. The Start and End variables point to the first and last observation in the Norm_times data set for that task (compare to Figure 4). Then we merge the index data set with the Jobs data set. For each matching observation we now go out to the Norm_times data set to read the right job time.

Note that it is often easier to perform an MxN merge using a full join in PROC SQL.

Program 4: Creating an MxN merge, i.e. each observation of one data set is matched to each observation of the other data set.

```
DATA Index(KEEP=Task Start End);;
RETAIN Start;
SET Norm_Times;
BY Task;
IF FIRST.Task THEN Start = N;
IF LAST.Task THEN DO;
   End = N;
   OUTPUT;
END;
RUN;
DATA MxN;
MERGE Jobs Index;
BY Task;
DO pointer = Start TO End;
   SET Norm_Times POINT=pointer;
   JobTime + Duration;
   OUTPUT;
END;
RUN;
```

Figure 7: Contents of index data set and result of MxN merge

Index data set

Obs	Start	Task	End
1	1	A	2
2	3	B	5

MxN result

Obs	Task	start	End	Subtask	Duration	Job Time
1	A	1	2	A1	10	10
2	A	1	2	A2	5	15
3	A	1	2	A1	10	25
4	A	1	2	A2	5	30
5	A	1	2	A1	10	40
6	A	1	2	A2	5	45
7	B	3	5	B1	20	65
8	B	3	5	B2	15	80
9	B	3	5	B3	10	90

AUTOMATIC RETAIN OF VARIABLES FROM DATA SETS

As mentioned in the section “the compilation phase” SAS retains all variables read from a data set automatically. This may cause a problem when using those variables for the result of calculations.
The following example illustrates this. Assume you have a data set LabTests, which contains a test ID, and a measurement. A second data set (WeightFactors) contains per test ID multiple weighing factors. These two data sets are printed in Figure 8:

Figure 8: Input data sets to demonstrate automatic retain

Data set: LabTests

Obs	Test ID	Measurement
1	A	34
2	B	41

Data set: WeightFactors

Obs	Test ID	Weight
1	A	1.0
2	A	1.5
3	A	2.0
4	B	2.0
5	B	4.0
6	B	8.0

The objective is to calculate the weighed measurements. In the following DATA step we use Measurement to contain the weighed measurements:

```
DATA WeightedResults;
MERGE LabTests WeightFactors;
BY TestID;
Measurement = Measurement * Weight;
RUN;
```

However, due to the automatic retain of Measurement the result is incorrect as you can see in the output in Figure 9

Figure 9: Erroneous calculation of weighed measurements due to automatic retain

Obs	Test ID	Measurement	Weight
1	A	34	1.0
2	A	51	1.5
3	A	102	2.0
4	B	82	2.0
5	B	328	4.0
6	B	2624	8.0

To avoid these problems you should use a new variable, in which case the original Measurement variable is the basis for all weighed measurement calculations.

THE UPDATE STATEMEMT

The purpose of the UPDATE statement is to transfer new data into a master file. The new data exist in a transaction file.

```
UPDATE MasterFile TransactionFile;
```

The way it works is similar to the way the MERGE statement works, however with a few fundamental differences:
The UPDATE statement always involves two data sets, the first is the master file, the second is the transaction file.
A BY statement *must* be used and each observation in the master file must have unique BY variable values.
Repeated values may occur in the transaction file, but that does not lead to repeated values in the output data set (the new master): just after the final observation of the BY group in the transaction file, an observation in the master is updated or appended.
Update only works for variables with a non-missing value in the transaction data set, in other words: missing values in the transaction data set are not transferred to the master data set.

EXAMPLE OF UPDATE

Assume that there is a data set with article numbers, descriptions and prices. We also have a data set with a number of updates to that file: a few prices have changed, some articles must be added and one description must be improved. (Figure 10).

Figure 10: Input data for UPDATE example

Data set: Articles

Obs	Art_Nbr	Description	Price
1	10020	plank 3x5	10.60
2	10030	plank 2x3	4.95
3	10050	U-profile steel	7.80
4	11030	chipwood 8mm1	2.95
5	11070	plywood18mm	65.50
6	20060	paint glos white	8.40
7	20100	paint acryl white	12.70
8	21090	paint brush 2x1	4.50
9	22001	abbrasive paper180	0.90
10	40090	electric drill	149.00
11	45030	handsaw	29.90
12	50010	screw 1x5	21.50
13	50030	nail 1.75	12.60

Data set: Updates

Obs	Art_Nbr	Description	Price
1	10050		8.15
2	11060	plywood 12mm	48.50
3	11060		48.95
4	20060		8.95
5	20105	acryl yellow	12.70
6	45030		24.90
7	50030	nail hard SP2 1.75	.

As you can see the Updates data set contains missing values. These will not be ported to the Articles data set. You also see 2 observations for the new article 11060. Only one will result. The following DATA step will update the Articles data set:

```
DATA RevisedArticles;
    UPDATE Articles Updates;
    BY Art_Nbr;
RUN;
```

The resulting RevisedArticles data set is presented in Figure 11.

Figure 11: Revised article data set, after UPDATE

Obs	Art_Nbr	Description	Price
1	10020	plank 3x5	10.60
2	10030	plank 2x3	4.95
3	10050	U-profile steel	8.15
4	11030	chipwood 8mm	12.95
5	11060	plywood 12mm	48.95
6	11070	plywood 18mm	65.50
7	20060	paint glos white	8.95
8	20100	paint acryl white	12.70
9	20105	acryl yellow	12.70
10	21090	paint brush 2x1	4.50
11	22001	abbrasive paper 180	0.90
12	40090	electric drill	149.00
13	45030	handsaw	24.90
14	50010	screw 1x5	21.50
15	50030	nail hard SP2 1.75	12.60

The new generation of the master data set is now called: RevisedArticles. In practice it is often inconvenient to give the data set another name. That can be resolved by working through ‘generation data sets’. First the old master is renamed to another name and the new one again gets the name Articles. This technique can be executed simply with the help of PROC DATASETS.

THE MODIFY STATEMENT

The MODIFY statement reads and modifies a data set ‘in place’. This means that no new data set is created in the DATA step as with SET, MERGE, or UPDATE, but the observation is modified directly in the original data set. One or two data sets can be declared in the MODIFY statement. If one data set is declared, the MODIFY statement works similarly to the SET statement, and otherwise more like the UPDATE statement. You mention the name of the data set both in the MODIFY statement and in the DATA statement. The options available are like those for the SET statement: POINT=.., NOBS=.., KEY=.., END=... The syntax can be summarized as:

```
MODIFY  dataset1 <dataset2> <options>;
```

MODIFY AS SET STATEMENT

Without any special options and with only one data set declared, the MODIFY statement reads the data set sequentially, each iteration of the DATA step a following observation, as with the SET statement. Because the MODIFY statement does not create a new data set, the same data set must be declared in the DATA statement as in the MODIFY statement and no new variables can be entered.

Should observations be deleted as a result of the programming of the DATA step, they continue to exist physically, but they get marked ‘deleted’ and can no longer be accessed. The next time the data set is rebuilt (for example in a DATA step with SET statement, or a PROC SORT), the ‘deleted’ observations are also physically removed.

The two DATA steps below perform the same function. The left DATA step modifies the data set in place, while the right DATA step first creates a new data set and then at completion of the step deletes the old one and renames the new one. MODIFY is in general (much) more CPU intensive, so you trade additional space needs (right step) for additional processing time (left step).

```
DATA In_Out;
   MODIFY In_Out;
   IF Count = . THEN Count = 0;
RUN;
```

```
DATA In_Out;
   SET In_Out;
   IF Count = . THEN Count = 0;
RUN;
```

The real advantage of the MODIFY statement comes in when you selectively rewrite observations: rewrite only the observations that have been updated. In many situations the use of MODIFY will then save time over the use of SET.

THE OUTPUT, REPLACE AND REMOVE STATEMENTS

In the above routines all observations were read from the data set and modified if so desired. At the end of the DATA step the observation is automatically written out, which means in case of MODIFY that the observation is updated with the new values for the variables.

You know that you can write observations to a data set selectively using the OUTPUT statement: it adds there and then an observation to the output data set, with the values that are present at that point in the DATA step. When using the MODIFY statement you have similar facilities, but with a choice of three statements. The OUTPUT statement adds a new observation to the end of the data set, the REPLACE statement replaces the value of the most recently read observation and the REMOVE statement removes the most recently read observation. Be careful that REMOVE is used here, not the DELETE statement. The DELETE statement interrupts the processing of an observation and outputs nothing. Thus the observation remains unchanged!
The following DATA step is a variant of the previous one:

```
DATA In_Out;
   MODIFY In_Out;
   IF Count = . THEN DO;
      Count = 0;
      REPLACE;
   END;
RUN;
```

In this step only those observations where the Count variable has been updated, are rewritten. As you can imagine this may save time. As a rule of the thumb, it is worth considering to use MODIFY in combination with REPLACE in stead of SET, if less than 5% of the observations need to be replaced.

UPDATING A MASTER FILE USING MODIFY

In the second form of the MODIFY statement two data sets are declared and there is a BY statement:

```
MODIFY masterdataset transactiondataset;
BY keyvariable(s);
```

The processing is now more or less similar to the UPDATE statement, although there are clear differences:
The data sets do not have to be sorted according to BY variable(s). The master data set must have been indexed on the BY variable(s).
Multiple observations with the same BY value may be present in the master data set.
Any variables that appear in the transaction data set but not in the master data set are not transferred.
New observations must be explicitly transferred to the master data set and are added at the end.

The MODIFY statement reads the transaction data set sequentially and looks for the observation related to it in the master data set. If this is found, then the non-missing values will be transferred from the transaction data set. If the master data set has multiple observations with the same BY values, then only the first is updated.
It is important to test the *IORC* variable: by doing so you can find out if the BY value you are looking for was present in the master data set. If not, the new value must be explicitly added using an OUTPUT statement. But using the OUTPUT statement has the additional consequence that also the existing observations must be explicitly updated with a REPLACE statement. The *IORC* variable and its use are described later.
Because the transaction data set is read sequentially, the modifications affect the master data set cumulatively.

KEY-ED ACCESS WITH MODIFY

Another variant of updating a master data set is the construction where you read a transaction data set with a SET statement and then use the KEY option in the MODIFY statement to access the right observation directly. Program 5 gives an example.
In this program there is a data set ScheduleData with schedule information about flights: one observation per flight/departure date. The data set is indexed with composite key “flight_key”, consisting of the flight number (variable: flight) and the departure date (variable: flight_date). It also contains, among other information, the aircraft type (variable AC_Type). The transaction file contains updates for the aircraft type of some departures. That data set is an "not-dated schedule file", i.e. it contains a first and last date on which the change is valid (variables: Effective and Discontinuation) and the days of operation (variable: days_of_ops, a subset of 1234567) on which the change is valid. Note that in the airline world day 1 is Monday.

The program reads the transaction data set sequentially and then for each observation it steps through all dates from Effective to Discontinuation, verifies that the flight_date fits in the list of days of operation and if so accesses the ScheduleData data set for the departure, updates the information and replaces the original observation. Note the use of the INDEX function. If the departure date matches with one of the days of operation the INDEX will return a value different from zero and therefore renders a TRUE value for the IF condition.

Program 5: Using the MODIFY and REPLACE statement to selectively update a master file.

```
DATA ScheduleData;
SET update_flights(RENAME=(ac_type = ac));
LENGTH ErrorMsg $40;
DO flight_date = effective TO discontinuation;
   IF INDEX(days_of_ops,PUT(WEEKDAY(flight_date-1),1.))
      THEN DO;
      MODIFY ScheduleData KEY=flight_key/UNIQUE;
      IF _IORC_ = 0 THEN DO;
         ac_type = ac;
       REPLACE;
       IF _IORC_ THEN DO;
            ErrorMsg = IORCMSG();
            PUTLOG ErrorMsg Flight Flight_date;
       END;
      END;
      ELSE PUT "Flight not in schedule" flight flight_date;
   END;
END;
RUN;
```

THE _IORC_ VARIABLE

When using the MODIFY statement or the SET statement with the KEY option, the SAS System defines a new automatic variable: _IORC_. This variable has the value 0 after a successful SET or MODIFY statement execution and the value -1 if an End-of-File condition occurred. All other values of _IORC_ indicate some error condition. There are two ways to inspect those errors. The IORCMSG function (no argument) translates the error number into a readable message, so it tells you what happened. In Program 5 you can see how to use this.

The autocall macro %SYSRC(...) works the other way around: you specify the type of error you want to test for as an argument and %SYSRC returns the corresponding value for _IORC_. Some common important conditions are listed in Table 1. In the source code of %SYSRC you can see the value of _IORC_ with various error codes. You should not hard code these values into your program. They could change in a later SAS release. %SYSRC would then change along with it. It is very important to test _IORC_, because the results of an unsuccessful SET or MODIFY are unpredictable. Program 6 contains an example how %SYSRC could be used.

Table 1: Some of the error conditions that can be tested using %SYSRC

_DSENMR	BY-value of observation in transaction data set not in master data set
_DSENOM	No corresponding observation in the master data set found.
_SENODEL	Observations may not be deleted from the data set
_SERECRD	No observation read from the input file.
_SWDLREC	Observation deleted

ADDING OBSERVATIONS WITH MODIFY

Program 6 shows a program similar to the one described in the paragraph on KEY-ed access using the MODIFY statement, but with a little twist: there may be new flights included in the transaction data set.
Figure 12 shows some observations from the ScheduleData and Update_Flights data sets.
The new flights should be added to ScheduleData and also to a separate file to inspect the added information.
The SAS Log will include information about the actions:

```
NOTE: There were 12 observations read from the data set WORK.UPDATE_FLIGHTS.
NOTE: The data set WORK.SCHEDULEDATA has been updated.  There were 232 observations rewritten, 53
observations added and 0 observations deleted.
NOTE: The data set WORK.NEWDEPARTURES has 53 observations and 12 variables.
```

Note the resetting of the _ERROR_ variable in the program. It has been mentioned before. This is done to prevent error messages from appearing in the log when a flight is not yet present in ScheduleData. If you would leave it on, all variables (including _ERROR_, _IORC_ and _N_) would be put out to the log after processing the observation, the standard action when _ERROR_=1.

Figure 12: Some observations in the input data sets for Program 6

Data set: ScheduleData

Obs	Flight	dep	arr	ac_type	flight_date	dept_time	arrv_time	capacity
…	…	…	…	…	…	…	…	…
100	HV5041	RTM	SXF	73G	05JUN06	19:45	21:00	149
101	HV5041	RTM	SXF	73G	06JUN06	19:45	21:00	149
…	…	…	…	…	…	…	…	…
8656	HV5131	AMS	BCN	73G	01JUL06	6:15	8:15	149
8657	HV5131	AMS	BCN	738	02JUL06	6:15	8:15	186
…	…	…	…	…	…	…	…	…

Data set: Update_Flights

Obs	Flight	days_of_ops	ac_type	effective	discontinuation
1	HV5131	135	738	01JUN06	01SEP06
2	HV5132	135	738	01JUN06	01SEP06
3	HV5131	47	767	01JUL06	01SEP06
4	HV5132	47	767	01JUL06	01SEP06
5	HV5133	123457	734	01JUL06	01SEP06
6	HV5134	123457	734	01JUL06	01SEP06

…

Program 6: Using the MODIFY statement to selectively update or add observations.

```
DATA ScheduleData NewDepartures;
   SET Update_Flights(RENAME=(ac_type = ac));
   DO flight_date = effective TO discontinuation;
      IF INDEX(days_of_ops,PUT(WEEKDAY(flight_date-1),1.)) THEN DO;
         MODIFY ScheduleData KEY=flight_key/UNIQUE;
         IF _IORC_ = 0 THEN DO;
            ac_type = ac;
            REPLACE ScheduleData;
         END;
         ELSE DO;
            IF _IORC_ = %SYSRC(_DSENOM) THEN DO;
               _ERROR_ = 0;
               ac_type = ac;
               OUTPUT;
            END;
         END;
      END;
   END;
RUN;
```

CONCLUSION

SET and MERGE belong to the most frequently used SAS DATA step statements. But don't forget UPDATE and MODIFY. Especially when dealing with a "transactions data set" to update a "master data set", UPDATE and MODIFY have much to offer. MODIFY can save processing time when only a limited number of observations needs to be updated, since it works "in place", meaning that observations that were not changed don't need to be written out again.

REFERENCES

Further information about SET, MERGE, UPDATE and MODIFY can be found in SAS 9.1 Language Reference: Concepts and SAS 9.1 Language Reference: Dictionary

RECOMMENDED READING

The basics of SET and MERGE can be found in "The Little SAS Book", by Lora Delwiche and Susan Slaughter. Several other books also provide useful information about these statements.
For more information about the functions that have been used in the examples refer to the Base SAS Users Guide.

CONTACT INFORMATION

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

Erik W. Tilanus
Synchrona
Horstlaan 51
3971 LB Driebergen-Rijsenburg
Phone: (+31) 343 517 007
Fax: (+31) 343 517 007
E-mail: erik.tilanus@planet.nl
Web: www.synchrona.nl

APPENDIX

In this appendix you can find the output of the data steps that have been presented in the introduction.

DATA STEP A.

Obs	ID	X	Y	Z
1	1	12	11	.
2	2	15	.	.
3	1	.	.	4
4	3	17	.	6
5	3	18	.	.

DATA STEP B.

Obs	ID	X	Y	Z
1	1	.	11	4
2	3	17	.	6

DATA STEP C.

Obs	ID	X	Y	Z
1	1	12	11	.
2	1	.	.	4
3	2	15	.	.
4	3	17	.	6
5	3	18	.	.

DATA STEP D.

Obs	ID	X	Y	Z
1	1	.	11	4
2	3	17	.	6
3	3	18	.	.

DATA STEP E.

Obs	ID	X	Y	Z
1	1	.	11	4
2	2	15	.	.
3	3	17	.	6
4	3	18	.	.

DATA STEP F.

Obs	ID	X	Y	Z
1	1	12	11	4
2	2	15	.	.
3	3	18	.	6

DATA STEP G.

Obs	ID	X	Z	Y
1	1	12	4	11
2	3	17	6	11
3	3	18	.	11

DATA STEP H.

Obs	ID	X	Z
1	3	17	6
2	3	18	.

DATA STEP I.

Obs	ID	X	Z
1	3	17	6
2	3	18	.

DATA STEP J.

Obs	ID	X	Y	Z
1	1	.	11	4

DATA STEP K.

Obs	ID	X	Z
1	3	18	