MWSUG 2017 – Paper SA10

The Building Blocks of SAS® Datasets – S-M-U (Set, Merge, and Update)

Andrew T. Kuligowski, HSN, St. Petersburg, Florida

Abstract / Introduction

S-M-U. Some people will see these three letters and immediately think of the abbreviation for a private university and associated football team in Texas. Others might treat them as a three-letter word, and recall a whimsical cartoon character created by Al Capp many years ago. However, in the world of the SAS® user, these three letters represent the building blocks for processing SAS datasets through the SAS DATA step. S, M, and U are first letters in the words SET, MERGE, and UPDATE – the 3 commands used to introduce SAS data into a DATA step.

This presentation will discuss the syntax for the SET, MERGE, and UPDATE commands. It will compare and contrast these 3 commands. Finally, it will provide appropriate uses for each command, along with basic examples that will illustrate the main points of the presentation.

SET Statement

According to the *Version 6 SAS Language Reference*, the **SET** statement "… reads observations from one or more existing SAS datasets." Under ordinary circumstances, each variable, on each observation, on each SAS dataset specified by the SET statement, is read into the SAS Program Data Vector, and each is subsequently written out to the new output SAS dataset - although this can be overridden with other logic in the DATA step.

The basic SET statement is very simple:

```
DATA  newdata;
   SET olddata;
   /* additional statements */
RUN;
```

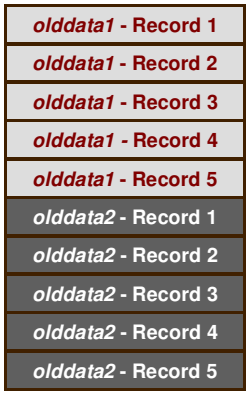
In this simple example, the SET statement introduces each record in SAS dataset *olddata* into the current DATA step. By default, those records are subsequently written out to SAS dataset *newdata,* although any or all of those records can be either output or discarded on a conditional basis if desired.

The SET statement is commonly used to <u>*concatenate*</u> two or more SAS datasets together. Let us look at a basic example:

```
DATA  newdata;
   SET  olddata1  olddata2;
   /* additional statements */
RUN;
```

In this example, the SAS Data Step will first read in each record from SAS dataset *olddata1* and, by default, write each of those records to SAS dataset *newdata*. Then, after the last record in *olddata1* has been processed, each record in *olddata2* will be read in and subsequently written out to *newdata*. By default, each observation in *newdata* will contain every variable in *olddata1* and every variable in *olddata2*. If there are variables that are defined in *olddata1* but not *olddata2*, the values will default to missing for those variables on the observations that are copied from *olddata1*. The opposite will be true for variables defined on *olddata2* but not on *olddata1*.

Depicted graphically, a single SET statement combines two or more SAS datasets <u>*vertically*</u>. Using the simple routine listed above, we would get:



There is an important condition that the SAS coder must be aware of when combining two or more SAS datasets. Variables that are common to two or more SAS datasets must have consistent definitions in each of those datasets in order to be used successfully in the SET statement (or, as will be discussed later, the MERGE or UPDATE statements). The most severe situation is when a variable is defined as character in one SAS dataset, but as a numeric in another. Should this occur, SAS will trip the internal _ERROR_ variable to 1 (for "true), set the return code to 8, and write the following ERROR message on the SASLOG:

```
ERROR: Variable variablename has
       been defined as both
       character and numeric.
```

If the common variables are both character or both numeric, but have different lengths, the output dataset will use the first length it encounters for that variable. This will be typically be in the first dataset in the SET statement. However, this default action can be overridden by inserting a LENGTH statement prior to the SET statement, as follows:

```
DATA  newdata;
   LENGTH commonvr $ 25.;
   SET  olddata1  olddata2;
   /* additional statements */
RUN;
```

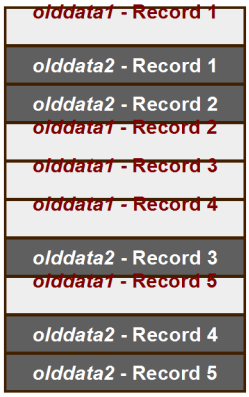
It should be noted that PROC APPEND can also be used to concatenate two SAS datasets, and it can often be more efficient than using a SET statement within a DATA step. However, PROC APPEND can only be used on two SAS datasets, while the SET statement can be used on three or more SAS datasets. In addition, the SET statement can be used in conjunction with other DATA Step statements to further massage the input data and to perform conditional processing; this additional functionality is not possible using PROC APPEND.

The SET statement can also be used to *interleave* two or more SAS datasets. In order to do this, each of the desired SAS datasets must contain the same key variables, and each must be sorted by those key variables. Then, the SET statement must be immediately followed by a BY statement listing those key variables, as follows:

```
DATA  newdata;
   SET  olddata1  olddata2;
   BY  keyvar1  keyvar2;
   /* additional statements */
RUN;
```

In the preceding example, *newdata* will contain the observations from both *olddata1* and *olddata2*. It will be sorted by the variable(s) referenced in the BY statement, in this case, *keyvar1* and *keyvar2*.

The addition of a BY statement causes the records to be interleaved, but does not alter the fact that the records are combined vertically. Graphically, this would look as follows:



It is also possible to use the SET statement to perform *direct access* (also known as *random access*) queries against a SAS dataset in order to retrieve a specific record. This is done by using the POINT= option on the SET statement. POINT= is followed by the name of a temporary SAS variable, which must have an integer value between 1 and the maximum record number in the SAS dataset being processed. This number can be easily obtained by using another option, NOBS=. NOBS= specifies a temporary variable that is populated during DATA step compilation with the number of observations in the applicable SAS dataset.

This concept is sometimes difficult to explain and to comprehend with only words; an example is needed for clarification. The following routine reads in every other record of a SAS dataset:

```
DATA  newdata;
   DO recno = 2 TO maxrec BY 2;
      SET  olddata  POINT=recno
           NOBS=maxrec;
      /* additional statements */
   END;
   STOP;
RUN;
```

Note that a STOP statement is required in order to terminate the current DATA step. This is because DATA step processing is concluded when the INPUT statement reads the End-of-File. However, when using the POINT= option, the INPUT statement never processes the End-of-File marker. Therefore, the DATA step is never terminated – not unless the coder specifically orders it via the STOP statement.

As stated earlier, NOBS= is populated at compile time, rather than during execution. This allows us to reference the number of records in a SAS dataset without actually executing the SET statement! The following example demonstrates this principle:

```
DATA  _null_;
   PUT maxrec 'Recs in olddata.';
   STOP;
   IF 1 = 0  THEN
      SET  olddata NOBS=maxrec;
RUN;
```

In this example, the SET statement defining the variable *maxrec* is coded AFTER the variable is to be displayed. Furthermore, the "IF 1 = 0" condition can obviously never be true, so the SET statement never actually executes. However, the variable *maxrec* is correctly populated with the record count of *olddata* during the compilation of the DATA step. This allows the record count to be printed, using the PUT statement, in the first line of the DATA step.

END= is another useful option. It specifies a temporary SAS variable that is normally set to 0 ('false'). However, it is "tripped" and reset to 1 ('true') when the INPUT statement encounters the last record in the SAS dataset

being read (or, if multiple datasets are specified, the last record in the last SAS dataset on the INPUT statement) This can facilitate any extra end-of-file processing that is required for the current DATA step.

```
DATA  newdata;
   SET  olddata  END=lastrec;
   /* additional statements */
   IF  lastrec = 1  THEN  DO;
      /* additional end-of-file */
      /* processing statements  */
      /* go here.               */
   END;
RUN;
```

There are a number of SAS Dataset options that can be used to enhance the processing of the SET statement. Dataset options must be enclosed in parentheses, and must immediately follow the dataset which they are describing. For example, a common request is to create a subset of a SAS dataset. This can be done by inserting an IF statement or a WHERE statement in the DATA step. However, it is also possible to do this by using the WHERE= dataset option in conjunction with the SET statement.

```
DATA  newdata;
   SET  olddata
        (WHERE=( keyvar < 10 ));
   /* additional statements */
RUN;
```

Dataset options can also be used to limit the number of records processed by the SET statement. FIRSTOBS= causes processing to start at a specified observation number, while OBS= causes processing to start at a specified observation number. They can be used together; the following example only processes the 1000th through the 2000th observations of a SAS dataset:

```
DATA  newdata;
   SET  olddata
        (FIRSTOBS=1000 OBS=2000);
   /* additional statements */
RUN;
```

There are several instances when it can be advantageous to use two or more SET statements within the same DATA step. For example, one coding technique is to store constants in a separate dataset, bringing them in at the beginning of a DATA step, as follows:

```
DATA  newdata;
   IF  _N_ = 1  THEN DO;
      RETAIN konst1-konst5;
      SET konstant;
      END;
   SET  olddata;
   /* additional statements */
RUN;
```

In this example, note that the SAS dataset *konstant* is only read during the first iteration of the DATA step, and that only one record is processed from *konstant*. The values that were read in from this record will be available throughout the entire execution of the DATA step, due to the RETAIN statement.

It is also possible to conditionally use the contents of one dataset to process another dataset, as in the following example:

```
DATA  newdata;
   SET  employee;
   IF  married = 'Y'  THEN
```

```
      SET  spouse;
   IF  childcnt > 0  THEN
      DO  i = 1 TO childcnt;
         SET  children;
      END;
   /* additional statements */
   /* go here.              */
RUN;
```

In this example, it is assumed that all 3 datasets are sorted by some common key variable. Following through the logic, it is assumed that every married employee has one and only one spouse, and the appropriate record is obtained. Furthermore, the routine determines the number of children that the employee has, and processes one record for each of them.
Depicted graphically, two or more SET statements combine SAS datasets *horizontally*. Basically, it would look as follows:

Dataset A - Record 1	Dataset B - Record 1
Dataset A - Record 2	Dataset B - Record 2
Dataset A - Record 3	Dataset B - Record 3
Dataset A - Record 4	Dataset B - Record 4
Dataset A - Record 5	Dataset B - Record 5

It should be noted that multiple SET statements could actually produce undesirable results. Since DATA step processing stops after the SET statement encounters an End of File marker – the *first* End of File marker in the case of multiple input datasets – the *smallest* dataset drives the number of iterations for the DATA step. In addition, the logic can get very cumbersome. There is an easier way to accomplish the same thing …

MERGE Statement

The **MERGE** statement " … joins corresponding observations from two or more SAS datasets into single observations in a new SAS dataset," to quote from the *Version 6 SAS Language Reference*.

The simplist example of the MERGE statement would be a *one-to-one* merge, as follows:

```
DATA  newdata;
   MERGE olddata1 olddata2;
RUN;
```

In this example, this routine takes the 1st record in *olddata1* and the first record in *olddata2*, and joins them together into a single record in *newdata*. This is repeated for the 2nd records in each dataset, the 3rd records, etc.

This technique is not often used in the real world, or more accurately, not often used intentionally. The main problem is that it is grounded in the assumption that there is a record-to-record relationship in each of the datasets to be merged, regardless of the contents of those records. This is not often the case.

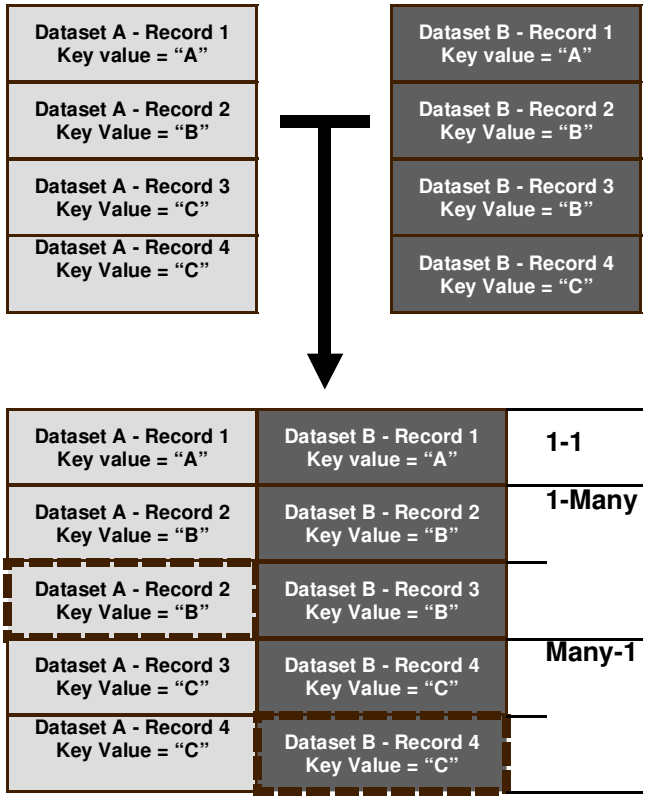
It is much more common to find that there is a record-to-record relationship based on the values of key fields found in both files. This approach, known as *match-merging*, requires all of the files in the MERGE statement to be sorted by the same variable(s). To show a simple example:

```
DATA  newdata;
   MERGE olddata1 olddata2;
   BY  keyvar(s);
RUN;
```

Match-merging is a very powerful tool. With two statements - MERGE, followed immediately by BY – the DATA step can handle cases of:

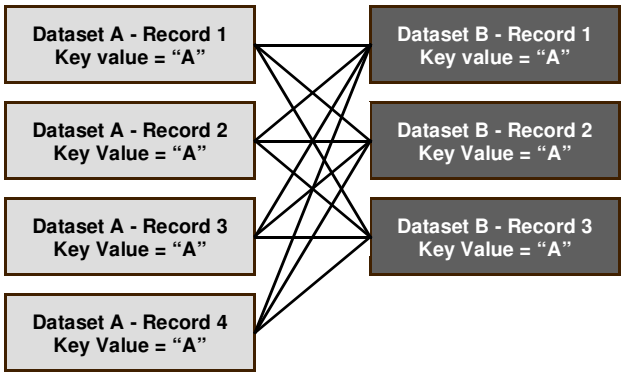
- One-to-one matching, where there is one record in each file containing the same key values.
- One-to-many matching, in which the first file has one record containing a particular set of key values and the second (or subsequent) file has multiple records containing those same key values, and
- Many-to-one matching, where the first file has multiple records containing a unique set of key values and the second file has only one record with those key values.

This is best shown graphically. The following example shows 2 datasets, both with 4 records each. Both files begin with a first record containing the same key value, illustrating 1-1 matching. The first file then contains one record with a different key value while the second file contains two records with that key, showing 1-many matching. Finally, the first file contains two records with another unique key, while the second file only has one record with that key, demonstrating many-1 matching:



It is also signicant to note what the example does not show. There are no examples of "1 to null" or "null to 1", or "many to null" or "null to many" merges, although these are all valid. These occur when a given key value exists in one file, but not the other. Each of these is a valid MERGE condition, however, and will result in record(s) being written to the output file. In this case, any variable that is only present on the file without the current key value will contribute null values for those variables to the final output dataset.

We do also not have any examples of many-to-many merges. This condition occurs when both files have multiple record with the same key values present.
To illustrate this graphically:



SAS cannot process many-to-many merges with the MERGE statement. They result in an ominous note to the SASLOG, and undesirable output results. However, be warned - the routine does NOT terminate with a non-zero condition code. In fact, processing continues on as though nothing is wrong, even though the results of the merge are almost definitely NOT what the author intended! Note that PROC SQL is capable of processing many-to-many merges. However, this is outside the scope of this presentation.

It is possible to programatically prepare your data to avoid the risk of attempting a many-to-many merge by removing multiple records with the same key values, using the NODUPKEY option on PROC SORT, as follows:

```
PROC SORT DATA=olddata1  NODUPKEY;
   BY  keyvar(s);
RUN;
```

However, this approach can also delete records that should actually be processed. The reader is encouraged to read this author's "Pruning the SASLOG…" presentation, as cited in the "References" section at the end of this presentation, for techniques to eliminate many-to-many merges.

Variable names that are common to two or more SAS datasets on the MERGE statement must have the same definition on each of those datasets, just like with the SET statement. However, unlike the SET statement, the value from the second dataset will overlay the value from the first dataset with MERGE. Sometimes this is desirable, while other times it is unwanted. In the latter case, the RENAME= option will allow the variables from *both* datasets to be written to the new output dataset, albeit with different names.

The use of the RENAME= option is simple:

```
MERGE olddata1
        (RENAME=(oldname=newname))
      olddata2;
```

The output dataset will contain the values of the variable *oldname* from both *olddata1* and *olddata2*, although the value from *olddata1* will be stored in the new variable, called *newname*.

It is possible to perform conditional processing, based on the source dataset for each record, by using the IN= dataset option. The IN= parameter creates a temporary variable associated with the dataset for which it is specified. The variable is set to 1 (for "True") if the specified dataset is contributing data to the current observation; otherwise it is set to 0. This allows for specialized processing, depending on the source of the current data. IN= can be used with the SET statement, but it is much more commonly found along with the MERGE statement.

Let us look at a simple example of IN=, with subsequent conditional processing:

```
DATA  newdata;
   MERGE olddata1(IN=in_old1)
         olddata2(IN=in_old2);
   BY  keyvar(s);
   IF in_old1 AND in_old2 THEN
```

```
      /* additional statement(s)*/
   IF in_old1 AND NOT in_old2 THEN
      /* additional statement(s)*/
   IF NOT in_old1 AND in_old2 THEN
      /* additional statement(s)*/
   IF NOT in_old1 AND
      NOT in_old2  THEN
      /* additional statement(s)*/
RUN;
```

The variable *in_old1* will be set to 1 (true) when the current observation in *newdata* is being fed from *olddata1* and 0 (false) when it is not. The same thing is true for variable *in_old2* and SAS dataset *olddata2*. The subsequent IF statements allow for special processing if the current observation contains data from both *olddata1* and *olddata2*, from *olddata1* but not *olddata2*, from *olddata2* but not *olddata1*, and from neither *olddata1* nor *olddata2*.

Of course, alert readers will quickly realize that the 4th and final IF statement is not necessary in the previous example. Since the DATA step is being fed from observations in *olddata1* and *olddata2*, there will never be an instance where the routine will be processing a record that is not present in either dataset! (It is sometimes beneficial to embed this explanation in a comment within your code, depending on your audience.) In addition, the experienced coder will quickly realize that this example could be made more efficient by using the ELSE statement; however, that is outside the scope of this presentation.

It should be noted at this point that most options available for the SET statement are also valid for use with MERGE, and vice verse. (Notable exceptions are POINT= and NOBS=, which are exclusive to the SET statement.) These topics are introduced in this paper under the command where the author has personally found them to be of most use in his daily activies.

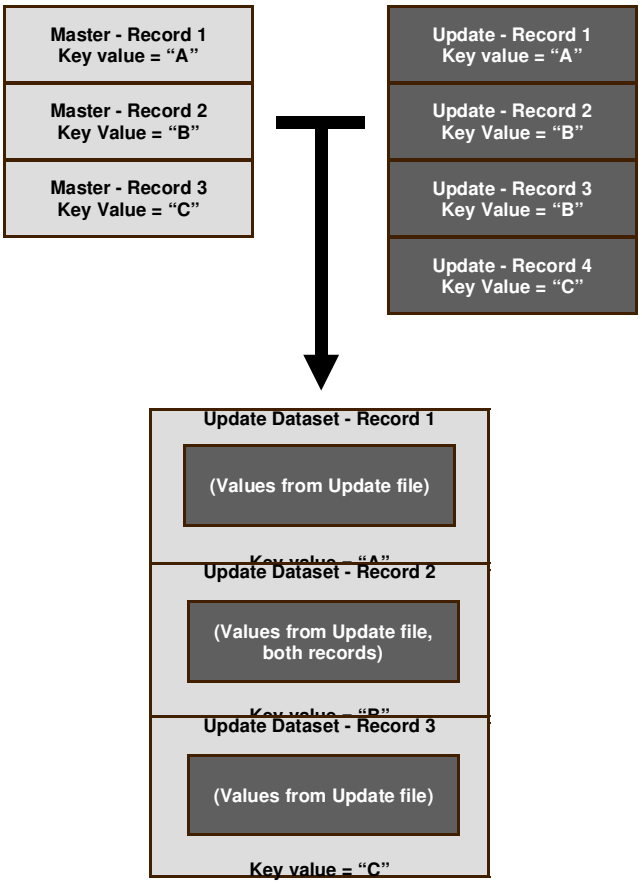
UPDATE Statement

The **UPDATE** statement is similar to the MERGE statement, "… but the UPDATE statement performs the special function of updating master file information by applying transactions …" to quote once more from the *Version 6 SAS Language Reference*.

The UPDATE statement combines records from two files in a horizontal fashion, like the MERGE statement. However, there are a number of significant differences between the two statements:

- UPDATE can only process two SAS datasets at a time – the *Master* dataset and the *Transaction* dataset. A single MERGE statement can process 3 or more SAS datasets.
- The BY statement is optional (although typically used) with MERGE, but is required with UPDATE.
- UPDATE can only process one record per unique BY group value in the Master dataset. It can process multiple records per unique BY group value in the Transaction dataset. However, in this case, each Transaction record is applied to the same record in the Master dataset, which means that transactions can be overlaid by subsequent transactions within the same DATA step.
- The UPDATE statement can avoid overlaying any given value in the Master dataset with a value in the Transaction dataset by setting the corresponding value in the Transaction dataset to missing. This would require more complex conditional logic to accomplish via the MERGE statement.

Depicted graphically, the UPDATE statement performs a modified version of a horizontal merge, in which values on the original records are overlaid with new information:



The UPDATE statement is simple to code – in fact, the only difference between it and the earlier MERGE statement / BY statement example is the word "UPDATE":

```
DATA  newdata;
   UPDATE olddata1 olddata2;
   BY  keyvar(s);
RUN;
```

However, the true utility of the UPDATE command becomes apparent after examining "before" and "after" sample data:

```
   SAS Dataset: MASTER
OBS    KEY1    UPDT1    UPDT2
 1      AL      1001       2
 2      FL      1002       4
 3      GA      1003       8
 4      MS      1004      16

    SAS Dataset: XACTION
OBS   KEY1  UPDT1  UPDT2  NEW3
 1     AL   1111     52     A
 2     GA   1133     54
 3     GA   2133      .     C
 4     MS      .      .     D
 5     LA   4555     65     E
```

```
     SAS Dataset: UPDATIT
OBS   KEY1  UPDT1  UPDT2  NEW3
 1     AL   1111     52     A
 2     FL   1002      4
 3     GA   2133     54     C
 4     MS   1004     16     D
 5     LA   4555     65     E
```

In this example, the first record in the Master File (for KEY1=AL) has its values changed for fields UPDT1 and UPDT2, as well as a value added for newly created field NEW3. The second record (KEY1=FL) is untouched, and NEW3 is set to missing. The third record (KEY1=GA) is actually changed twice due to two separate records in the Transaction dataset, with only the final set of changes stored to the output file. The fourth record (KEY1=MS) does not have its original two values adjusted, because the values are set to missing in the transaction dataset. However, a value is inserted for the new variable NEW3. The last record in the Transaction file (KEY1=LA) does not exist in the Master dataset, so it is added.

The question arises – what if you WANT to replace an existing value with a missing value? This is possible, but requires a little extra coding. The MISSING statement, added to the DATA step, will define special missing values that can be used in the Transaction data. The values "**A**" through "**Z**" will display as coded, while an underscore "_" will invoke the standard missing data representation of a dot "**.**". Note that the MISSING command must be present during both the creation of the transaction data and in the UPDATE DATA step.

MODIFY Statement

The **`MODIFY`** statement was added in Version 6.07 of the SAS System. It has many of the same capabilities of the SET, MERGE, and UPDATE statements – with one important difference. To quote from *SAS Technical Report P-222*, the MODIFY statement "… extends the capabilities of the DATA step, enabling you to manipulate a SAS data set in place without creating an additional copy."

Neither SET, nor MERGE, nor UPDATE has the ability to update a dataset in place. They give the appearance of doing that when the DATA statement specifies a dataset name that is the same as one in SET, MERGE, or UPDATE. However, in actuality, the DATA step creates a new dataset in this instance, and then replaces the old dataset upon completion. The MODIFY statement avoids the creation of this temporary dataset, along with the extra temporary disk storage that it requires and the time – processing and clock – it takes to make it.

However, as one might expect, there is a trade-off that must be considered before using the MODIFY command. To quote from the manual: ***"Damage to the SAS data set can occur if the system terminates abnormally during a DATA step containing the MODIFY statement."*** The user must take special care to prevent their dataset from being corrupted while being MODIFY-ed, whether the problem is caused by the execution of buggy code or from the careless absence of a UPS upon a power outage.

The most basic form of the MODIFY statement is quite simple:

```
DATA  dataset;
   MODIFY dataset;
   /* additional statements */
RUN;
```

It acts much like the SET statement discussed earlier. However, there are some major differences between the two:

- As mentioned above, the MODIFY statement causes the current dataset to be changed in place -without expending the I/O and disk space to generate a new dataset. Therefore, as should be expected, the name of the output dataset *must* be the same as the one being modified.

- Additional variables cannot be added and unneeded variables cannot be deleted from a MODIFY-ed dataset. It should be noted that no ERROR or WARNING statement will be generated if these statements are present in the SAS code; the requested structure alterations will simply not be applied.

- The **REPLACE** statement causes the current record to be rewritten in the MODIFY-ed SAS dataset with any changes applied. The **OUTPUT** statement causes a new record to be written to the MODIFY-ed dataset – however, the original record will still be present on the output file, resulting in two separate records upon completion of the DATA step. This difference cannot be sufficiently emphasized – the OUTPUT statement, when used with MODIFY causes a second, duplicate record to be added to the end of the current SAS dataset. The user must be certain not to use OUTPUT instead of REPLACE, which changes the current record in place and does not result in an additional record.

- You cannot DELETE an observation when using the MODIFY statement. You can, however, use the **REMOVE** command to eliminate unwanted observations.

The MODIFY statement can be used for sequential (random) access. The syntax is similar to that of the SET statement as discussed earlier:

```
DATA  dataset;
   DO recno = 2 TO maxrec BY 2;
      MODIFY  dataset  POINT=recno
              NOBS=maxrec;
      /* additional statements */
   END;
   STOP;
RUN;
```

The need for the STOP statement is also similar to that of the SET statement - assuming the coder wishes to avoid the pitfalls of an infinite loop.
It is also possible to join two or more SAS datasets using the MODIFY statement. As with the other examples in this section, the syntax is similar to that of the SET and UPDATE statements.

```
DATA  master;
   MODIFY  master  transact;
   BY  keyvar1  keyvar2;
   /* additional statements */
RUN;
```

Both the master and transaction datasets must contain the same key variables. However, they do *not* need to be sorted by those variables – the presence of a BY statement causes the SAS System to invoke a dynamic WHERE clause. Please note that, although not required, it is highly recommended for efficiency sake that the datasets be sorted or indexed by the key variables in the BY statement.)

There is one other important difference - multiple records with the same key values act differently when processed with the MODIFY statement. Duplicate key values in the master file will *not* be altered – only the first record of occurrence is updated due to the aforementioned WHERE clause processing. (As one might expect, multiple records in the transaction file will overwrite each other, so that only the changes in last transaction record in the series will be available in the master dataset at the end of the DATA step.) These differences provide a "safety valve" to prevent unwanted alterations to your permanent SAS data.

Speaking of "safety valves", the user is highly encouraged to incorporate the automatic variable _IORC_ into their routines. _IORC_ contains the return code for every I/O operation that is performed by the MODIFY statement – or rather, that for each one that is attempted. The simplest use would be to simply ensure that the field contains a zero before continuing on with the routine. It can be made more complex, with logic handling specific errors, or by using the IORCMSG function to obtain and display the associated error message for the return code. (Note that IORCMSG is not available under Version 6 of the SAS System.)

Conclusion

This presentation is designed to be a brief introduction to the SET, MERGE, UPDATE, and MODIFY commands. It is not a "shopping list" of the various options available for each command; that information is readily available in the manuals, as listed in "References" below. The reader is encouraged to sit down at the computer and try examples of each command to facilitate his or her learning of the subject; only after hands-on trial will the information truly be meaningful to the reader.

APPENDIX A
LIBNAME Statement

It has been brought to the attention of the author that some readers of this presentation may be unfamiliar with the `LIBNAME` statement. Since this statement or its equivalent is required in order to permanently store SAS data, this section has been included as a reference for any who may need it.

All SAS datasets are stored in SAS Data Libraries. In fact, the standard specification for a SAS dataset contains two levels – the SAS Data Library name, and the individual SAS dataset name, separated by a period, or "dot" if you prefer. Of course, even the newest of SAS users will realize that many SAS datasets are represented by only one name. This is because the default SAS data library is "WORK". WORK is automatically defined when SAS is invoked, and the absence of a Data Library name in a SAS dataset name causes SAS to use the WORK library.

Each SAS Data Library must be denoted by a LIBNAME. A LIBNAME, also commonly known as a LIBREF, is a shorthand representation, or nickname if you prefer, of the actual dataset name as defined by your particular operating system.

There are several ways to define a LIBNAME to the SAS System. One of them is to provide a file reference to the external file using a host system command outside of the SAS System - a JCL "DD" statement under IBM's MVS, for example. To cite an example that has already been referenced, this is the method by which the WORK library is allocated to the SAS session.

An alternate method is to use the **LIBNAME** statement under SAS. The LIBNAME statement is a global statement, and is executed outside of the DATA step. The generic syntax for this statement is:

```
LIBNAME libref  <engine>
        'external file' <options> ;
```

"Engine" and "options" will not be discussed at this time; they will be deferred to a more advanced presentation. As one might expect, however, many of the options for the LIBNAME statement are dependent on the operating system. Consult the appropriate "Companion" document for your operating systems for details.

It is also worth noting that the LIBNAME statement can be executed with two special keywords under any operating system:

- **CLEAR** will remove a reference to an existing SAS Library:
  ```
  LIBNAME libref CLEAR;
  ```
- **LIST** will write the attributes of the specified LIBNAME to the SASLOG:
  ```
  LIBNAME libref LIST;
  ```

There is also a **LIBNAME** *function* that can be invoked from within a DATA step to perform the same purpose. The syntax is similar to the LIBNAME statement:

```
LIBNAME(libref, 'external file',
        ,<engine>, <options> );
```

APPENDIX B
References / For Further Information

Kuligowski, Andrew T. (1999), "Pruning the SASLOG – Digging into the Roots of NOTEs, WARNINGs, and ERRORs". *Proceedings of the Seventh Annual Conference of the SouthEast SAS Users Group*. USA.

Riba, S. David. "The SET Statement and Beyond: Uses and Abuses of the SET Statement". *http://www.jadetek.com/download/jade_set.pdf*

SAS Institute, Inc. (1990), *SAS Language: Reference, Version 6, First Edition.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (2000), *SAS OnlineDoc, Version 8.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1994), *SAS Software: Abridged Reference, Version 6, First Edition.* Cary, NC: SAS Institute, Inc.

SAS Institute, Inc. (1991). *SAS Technical Report P-222, Changes and Enhancements to Base SAS Software, Release 6.07.* Cary, NC: SAS Institute, Inc.

Special thanks to Dave Riba and Ian Whitlock for their suggestions while preparing this paper.

The author can be contacted via e-mail as follows:
KuligowskiConference@gmail.com