Multiple Set Statements in a Data Step:
A Powerful Technique for Combining and Aggregating Complex Data

Renu Gehring

SAS Instructor
Ace-Cube, LLP
Beaverton, OR

Health Care Analyst
CareOregon, Inc.
Portland, OR

ABSTRACT

The use of multiple set statements in a data step is little known and often misunderstood. This is because two or more set statements in a data step can lead to loss of data and incorrect results. However, when used with additional code elements, multiple set statements prove to be powerful in combining and aggregating complex data effectively. This paper explains how multiple set statements work and then describes several applications, including one involving allocation of healthcare revenues.

KEYWORDS: SAS® Data Step Processing, Program Data Vector, Data Aggregation, Accumulator Variables, *Merge, Set, Do Loops, Nobs and Point* options.

INTRODUCTION

The data step language provides a rich array of methods to manipulate, combine, and aggregate data. Unlike procedures which are mostly pre-programmed routines, the data step language is chockfull of elements that can be used for flexible and creative programming. Multiple set statements in a data step are one example of the power of the data step. But multiple set statements can also be misused easily, leading to loss of data and incorrect results. However, when used with additional code components and a thorough understanding of data step processing, multiple set statements prove valuable in effectively combining and aggregating complex data. This paper explains how multiple set statements work and then describes several applications, including one involving allocation of healthcare revenues.

1: Syntax and Usage of the Set Statement

The syntax of the set statement is both easy and straightforward, but the tasks accomplished by slight variations in its syntax are complex. This section describes the different ways to use the set statement.

1.1: The Set Statement: Basic and Absolutely Essential

Before diving into the complexities of data and syntax, let us bone up on some basic SAS rules regarding the set statement and the data step. The data step is used to manipulate data and place it in a new data set. The set statement followed by the name of an existing SAS data set simply specifies the source data set that is read. In the following code snippet, a data set called *DataB* is created and its source data set is *DataA*.

```
data DataB;
   set DataA;
   *Manipulate Data here;
run;
```

The set statement can be used concatenate or stack data sets vertically. In the following code, *Big* is simply a vertical stack of *Small1* and *Small2*. So we know that the set statement followed by the names of two or more data sets creates a tall stack of data.

```
Data Big;
   set Small1 Small2;
   *Manipulate Data here;
run;
```

In addition to creating a vertical stack, the set statement can be used to interleave data sets by an identifying variable. Consider the following data sets *A* and *B*.

Data Set A

Name	Age
Amy	32
Jake	28
Tom	55

Data Set B

Name	Age
Ben	36
Mary	35

Stacked Data Set A and B

Name	Age
Amy	32
Jake	28
Tom	55
Ben	36
Mary	35

Both *A* and *B* have the same variables, Name and Age. A set statement followed by the names of the data sets *(set A B*) produces a vertical stack (Stacked data set *A* and *B*). This stack is not sorted by name. In order to interleave these data sets by name, we first sort the two data sets by name and then use a set statement and a by statement.

```
*Interleaving;
data BigSorted;
  set A B;
  by name;
run;
```

Data Set BigSorted

Name	Age
Amy	32
Ben	36
Jake	28
Mary	35
Tom	55

Interleaving is useful when combining several large data sets because the end product is a sorted data set. By sorting several smaller data sets you avoid sorting the combined data set, which can be an important advantage when facing space and memory constraints.

1.2: How Multiple Set Statements work (or don't)?

Multiple set statements in a data step can be tricky. Why? Take the following two data sets and try submitting this code snippet.

```
data Overwrite;
  set A;
  set B;
run;
```

Data Set A

Name	Gender
Amy	F
Jake	M
Tom	M

Data Set B

Name	Age
Ben	36
Mary	35

Data Set Overwrite

Name	Gender	Age
~~Amy~~ Ben	F	36
~~Jake~~ Mary	M	35

What just happened here? Ben is 36 years old, but clearly not a female. Mary is 35 years old, but not a male. And what happened to the other three rows of data?

When SAS finds two or more set statements in a data step, this is how it executes the data step:

1. Reads the first observation from *A* and places it into an area of memory called the Program Data Vector (PDV). The PDV also contains two useful automatic variables, _N_ and _Error_. The former tracks the number of iterations of the data step and latter takes on a value of 1 if any data errors occur. If we could peek into the PDV this is what we would see:

_N_	_Error_	Name	Gender
1	0	Amy	F

2. Reads the first observation from *B* and also places it into the PDV. But here comes the tricky part. SAS overwrites values of common variables from *A* with the new values from *B*. In our case, the common variable is name and its value is overwritten with values from *B*. Hence the value 'Amy' is overwritten with the value 'Ben'. The age variable is added to the PDV. And the observation is outputted to *Overwrite*. This ends the first iteration of the data step. The PDV would look like this:

_N_	_Error_	Name	Gender	Age
1	0	Ben	F	36

3. A similar logic is followed in the second iteration of the data step. SAS reads the second observation from *A* and places it into the PDV. Then SAS reads the second observation from *B* and overwrites values of common variables. Hence Mary is assigned to the male gender.
4. At the end of the second iteration of the data step, SAS stops processing. This is because SAS encounters the end of file marker on the smallest file. This explains why our third row went AWOL.

If all multiple set statements do is over-write data and shortchange us on the number of records, why use them? Consider a parallel universe of data called "Data So Simple It Can't Be Real". In this world, you have two very large files that share a unique identifier, ID. These files have an exact one to one match, which means that every ID in the first file has its match in the second file and vice versa. In addition, these files are sorted by ID. This means that that 8,765,875th record in the first file and the second file will have information for the same ID. Your goal is to merge the files and the first line of attack is the *merge* statement. Due to its memory and disk space gobbling feature, the *merge* statement fails. In this case, multiple set statements come to your rescue, combining your two files efficiently and accurately.

How often does this happen in our real universe of complex and messy data? Suffice it to say that in many years of SAS experience, I am still on the lookout for this perfect application of multiple set statements.

So how can we use multiple set statements while avoiding the pitfalls of losing data and accuracy? We need additional code elements, such as the *do loop*, the *point* option, and the humble *if-then* logic. And to show you the true power of multiple set statements, we need a data structure on which *merge* would not produce accurate results.

2: Applications of Multiple Set Statements

This section details two applications of the multiple set statements. The first involves combining data sets that have a many to many relationship and is an illustrative example for demonstrating the syntax involved. The second application involves allocation of

monthly health care revenues. By combining and aggregating complex data, this example shows the true power of multiple set statements.

2.1: Combining Data Sets with a Many to Many Relationship

As a SAS instructor, I like to use data that is as close to real data as possible. I have created mock data sets based on Oregon Hospital Discharge Data. I show key concepts of data analysis and SAS programming with *Discharge*, which contains inpatient discharge data, including dates of admission and discharge, and several codes that serve as look ups to other data sets. One of these look up or reference data sets is *Payer*, containing insurance carrier data for each patient. As *Discharge* allows for readmissions and one patient may have more than one insurance carrier, a complex relationship exists between *Discharge* and *Payer*. As an example consider the following discharge records:

Data Set Discharge

ID	Date of Admission	Date of Discharge
1	10/15/ 2008	10/19/ 2008
1	11/01/2008	11/05/ 2008

Data Set Payer

ID	Insurance Provider
1	Medicare
1	Private HMO

Discharge shows two discharge records for patient 1 because this patient was hospitalized twice. *Payer* indicates that this patient was covered by both Medicare and a Private HMO. Let us assume that Medicare was the patient's primary insurance carrier whereas HMO was his secondary insurance provider. This means Medicare and the Private HMO will be responsible for some portion of the charges that the patient accrues on each of his inpatient stays. In data terms, this means that each record *Discharge* should be combined with every record from *Payer*. The complex relationship exhibited by *Discharge* and *Payer* is called a many to many relationship by data modelers. The following table shows what a combined version of *Discharge* and *Payer* would look like.

Discharge and Payer Combined

ID	Date of Admission	Date of Discharge	Insurance Provider
1	10/15/2008	10/19/2008	Medicare
1	10/15/ 2008	10/19/2008	Private HMO
1	11/01/2008	11/05/2008	Medicare
1	11/01/2008	11/05/ 2008	Private HMO

As a billing specialist who has dabbled in SAS code, you submit the following code hoping for the correct results.

```
data DischargePayer;
  merge Discharge Payer;
  by id;
run;
```

You get a cryptic note in the SAS Log and the following result:

```
NOTE: MERGE statement has more than one data set with repeats of BY
values.
NOTE: There were 2 observations read from the data set WORK.DISCHARGE.
NOTE: There were 2 observations read from the data set WORK.PAYER.
NOTE: The data set WORK.COMBINE has 2 observations and 4 variables.
```

Discharge and Payer Combined with the *Merge* Statement

ID	Date of Admission	Date of Discharge	Insurance Provider
1	10/15/2008	10/19/2008	Medicare
1	11/01/2008	11/05/2008	Private HMO

The *merge* statement failed to produce the correct results due to two important SAS rules.

1. In a data step, SAS reads observations sequentially.
2. Once an observation is read into the PDV, it is never re-read.

Because of these rules, the *merge* statement should not be used with data sets that have a many to many relationship.

Where the *merge* statement fails, multiple set statements combined with other syntax in a data step succeed. Consider the first iteration of the data step code below. The first observation from *Discharge* is read into the PDV❶. A loop (*i=1 to num*) iterates once for every observation in *Payer*❷. The set statement with its *rename, point,* and *nobs* options does a variety of tasks❸. First, the *nobs* option gathers the number of observations in *Payer*, feeding the value to *num*. (*Num* is equal to 2 because *Payer* has two observations.) The *i*th observation of *Payer* is retrieved by direct access. This is accomplished by setting the *point* option to equal *i*. To prevent over writing of *id* from *Discharge* with the *id* from *Payer*, the *rename* option renames *id* to *idnum*❸. The *if..then* logic links observations from both *Discharge* and *Payer* by *id* and sends the matched pair to the output data set, *CorrectMerge*❹. At this point, the loop iterates again, with *i=2*. The PDV still contains the first observation of *Discharge* and the second iteration of the loop attaches the second observation of *Payer* to the existing observation in the PDV to achieve a Cartesian product.

```
data CorrectMerge(drop=idnum);
  set Discharge;  ❶
  do i=1 to num;  ❷
    set Payer(rename=(id=idnum)) nobs=num point=i; ❸
    if id=idnum then output;   ❹
  end;
run;
```

Here is a snapshot of the PDV after the loop has iterated once.

ID	Date of Admission	Date of Discharge	Insurance Provider	Idnum
1	10/15/ 2008	10/19/ 2008	Medicare	1

Here is what the PDV would look like after the second iteration of the loop.

ID	Date of Admission	Date of Discharge	Insurance Provider	Idnum
1	10/15/ 2008	10/19/ 2008	Private HMO	1

After the data step has completed, the results would be as follows:

CorrectMerge Data Set

ID	Date of Admission	Date of Discharge	Insurance Provider
1	10/15/2008	10/19/ 2008	Medicare
1	10/15/2008	10/19/ 2008	Private HMO
1	11/01/2008	11/05/ 2008	Medicare
1	11/01/2008	11/05/ 2008	Private HMO

Although the results are correct, multiple set statements as applied above seem overkill, especially since the same results can be achieved by the SQL procedure with relatively simple code. Where the true prowess of multiple set statements comes into play is in combining **and** aggregating complex data. The next application deals with complex data that needs to be aggregated before a complex combination occurs.

2.2: Allocation of Health Care Revenue

A health insurer receives payments to cover the health care costs of members of a large entity. Payments are received monthly and are dependent, to a large extent, on the individual member's health status or score. Payments for sicker members are higher than payments for relatively healthy members. As members can become sicker or healthier over time, payments can change from month to month. Although updates to health status are recorded continuously, current payment can often be based on a prior health score. This leads to retroactive payments, most of which occur at fixed intervals within one to two years from the month(s) of service. These adjustments to the original payments can be negative in the case of the member who becomes healthier over time and positive for members whose health status deteriorates over time. Consider the following payment history.

Mock Payment History

ID	Start of Service	End of Service	Adjusted Payment	Payment
1	01/01/2008	01/31/2008	No	$1,500
1	02/01/2008	02/29/2008	No	$2,500
1	01/01/2008	02/28/2008	Yes	$2,000

According to the above data, three payments covering the months of January and February 2008 were received. The first two payments represent original payments and are different to illustrate a change in the member's health score. The member became sicker in January after January's original payment had been made. The change in health status resulted in an update to her health score which was the basis for the higher original payment for February. The third payment, or the adjusted payment, reflects another update to the member's health score. The member's health deteriorated for the second time in February. Because the member's most recent health status determines her payments for the previous months of the year, the adjusted payment is applicable to both January and February.

What payment was received for this member in January? Is it $1,500? Even a cursory glance at the data reveals that $1,500 is not the only payment received for January. Because the adjusted payment is applicable to January as well as February, part of the adjusted payment of $2,000 should be counted as the payment for January. But what part? Should half of the adjusted payment of $2,000 be applied as the payment for January? If this is correct, then the January payment is $2,500 (the original payment of $1,500 plus half of the adjusted payment of $2,000).

It turns out that $2,500 is the wrong answer. It is incorrect because the member's final health status in a calendar year is used to determine her retroactive payments for all previous months of the year. In this case, the member's most recent health status in February determines her retroactive payment for the months of January and February. As a result, the adjusted payment of $2,000 is not allocated equally between January and February.

Mathematically, what this logic implies is that we add both the original and the adjusted payments for the entire two months and then divide by two. Programmatically, the calculation is accomplished in two discrete steps. First, we separate payments by type, placing adjusted payments in *Adjusted* and original payments in *Original*. Second, for each adjusted payment, we combine the stream of original payments that relate to the adjusted payment's time period. Because multiple set statements are capable of aggregating and combining complex data, they are perfect candidates for this task.

Step 1: We begin by siphoning off original and adjusted payments to their separate data sets.

Data Set Original

ID	Start	End	Payment
1	01/01/2008	01/31/2008	$1,500
1	02/01/2008	02/29/2008	$2,500

Data Set Adjusted

ID	Start	End	Payment
1	01/01/2008	02/29/2008	$2,000

Step2: Next we combine the stream of original payments that relate to the time period of each adjusted payment and add the resultant sum to the adjusted payment.

Consider the accompanying SAS code. As the data step begins its first iteration, the set statement reads an observation from *Adjusted* and places it into the PDV❶. The SAS data step generally iterates once for every observation and observations are read sequentially. Since we have one observation in *Adjusted*, the data step will iterate once. *Payment2*, an accumulator variable whose function is to add the relevant original payments, is created❷. Note that *payment2* is set equal to 0. This is important because after every iteration of the data step, the value of *payment2*, a placeholder for the aggregated sum of original payments, must be wiped out. Values of accumulator variables are retained across iterations of the data step; hence the manual initialization wipes the slate clean. A loop (*i=1 to num*) will iterate once for every observation in *Original*❸. The set statement along with its options and *rename* statement performs a variety of tasks❹. First, the *nobs* option gathers the number of observations in *Original*, feeding the value to *num*. (*Num* is equal to 2 because *Original* has two observations.) The *i*th observation of *Original* is retrieved by setting the *point* option equal to *i*. To prevent over writing, the variables *start*, *end*, *id*, and *payment* are renamed to *startnum*, *endnum*, *idnum*, *opayment* respectively❹. The *if..then do* loop checks whether the date of the original payment falls between the beginning and ending dates of the adjusted payment❺. If so, the value of *payment2* increments by the value of *opayment*❻. The next loop checks whether the ending date of the last original payment record read is the same as the ending date of the adjusted payment record❼. *Totpayment*, the sum of original stream of payments and the adjusted payment, is created❽. In a separate data step (not shown), *totpayment* can be divided by the number of months (2 in this case) to derive at the correct computation of monthly payments.

```
Data combine;
   set adjusted;  ❶
      payment2=0; ❷
      do i=1 to num;❸
         set original (rename=(start=startnum end=endnum
         id=idnum payment=opayment)) nobs=num point=i; ❹
            if id=idnum and startnum>=start and
            endnum <=end then do;  ❺
               payment2+opayment;  ❻
               if end=endnum then do;  ❼
                  totpayment=payment2+payment;  ❽
                  output;
               end;
            end;
       end;
```

Here is a snapshot of the PDV after the loop has iterated once.

ID	Start	End	Pay-ment	Id-num	Start-num	End-Num	O-Payment	Payment2	Tot-Payment
1	01/01/08	02/28/08	2,000	1	01/01/08	01/30/08	1,500	1,500	3,500

Here is what the PDV would look like after the second iteration of the loop.

ID	Start	End	Pay-ment	Id-num	Start-num	End-Num	O-Payment	Payment2	Tot-Payment
1	01/01/08	02/28/08	2,000	1	02/01/08	02/28/08	2,500	4,000	6,000

As can be seen, the monthly payment for January 2008 is $3,000 as is the February 2008 payment. The adjusted payment of $2,000 is not distributed equally between the two months. Rather, a lion's share $1,500 of $2,000 goes toward January and only $500 is allocated to February.

The above application demonstrates the prowess of multiple set statements and additional code components. In a relatively compact data step, we accomplish both a combination and an aggregation of data, while implementing complex logic.

It turns out that the above example is highly simplified. This is because the real data is full of complexities such as multiple adjusted payments in a calendar year, adjustments that have overlapping time periods, or my personal favorite, adjustments made for the months of service when no original payment was received. In order to resolve these issues, multiple set statements had to be combined by a complex macro which is beyond the scope of this paper.

CONCLUSION

Applications of multiple set statements abound in a variety of industries. I conclude by providing another application of multiple set statements and challenge readers to provide a solution utilizing multiple set statements. A large utility company is interested in evaluating the efficacy of energy saving programs. Utilization patterns need to be

controlled for outside factors such as weather. In order to control for weather, it is necessary to attach aggregate weather information, for example, the number of extremely hot and cold days, to each customer's monthly utilization data. As utilization metrics are contained in customer billing data, the programmer's task consists of marrying billing data with daily weather data. The combination of billing data and weather data is complicated because of two factors. First, the billing data is large containing millions of observations across several years. Second, dates of utilization vary for each customer because the meter information is recorded on different days. This can be carried out effectively and efficiently with multiple set statements and the solution is left to the reader.

This paper has demonstrated that multiple set statements combined with additional code components prove useful in combining and aggregating complex data effectively. The paper's step by step explanation of SAS data step processing and code components serves to simplify the non-intuitive and difficult syntax that often accompanies multiple set statements. The hope is that more SAS programmers will begin to start taking advantage of multiple set statements and put their power to work.

ACKNOWLEDGMENTS

Special thanks in reviewing this paper go to Aaron Winkel of CareOregon, Inc and Mark Thompson of ForeFront Economics, Inc.

CONTACT INFORMATION

If you have any questions regarding this paper, please contact:

Renu Gehring
Ace-Cube, LLP
17715 NW Fall Ct
Beaverton, OR 97006
Phone: 971-235-5782
Email: Renu.Gehring@ace-cube.com

CareOregon, Inc
315 SW Fifth Avenue
Portland, OR 97204
Phone: 503-416-5754
Email: gehringr@careoregon.org