INTRODUCTION

Most SAS programmers are familiar with the following uses of the SET and MERGE statements for processing SAS data sets:

1) To read a SAS data set sequentially, one observation at a time.

2) To concatenate two or more SAS data sets.

3) To interleave two or more SAS data sets.

4) To perform a ONE TO ONE MERGE.

5) To combine observations from two or more data sets into one observation.

This paper will build upon these standard uses of the SET and MERGE statements to demonstrate solutions to complex programming problems. The problems have been simplified for the purposes of illustration in this paper. The concepts behind these examples make the power of the SAS programming language evident in controlling the processing of data in the data step. Some of the concepts presented in the paper entitled 'THE SAS SUPERVISOR' (reference 1) are reviewed through practical examples. The sample problems will include a discussion of the END OF DATA STEP FLAG (EDSF) and the INITIALIZE TO MISSING VECTOR (ITMV). While building on the concepts outlined in reference 1, this paper will reinforce the capabilities of the SET and MERGE statements with respect to the following:

1) Every SET statement in a data step activates its own READ pointer.

2) The FIRSTOBS option on a SET statement can be used to perform a LOOK AHEAD MERGE.

3) The IN = variables can be reset by the programmer.

4) The POINT option on a SET statement can be used to read a specific observation, or group of observations.

5) The NOBS option on a SET statement is available for use at COMPILe time.

6) A JOIN, or ALL COMBINATIONS MERGE can be achieved by creating POINTER data sets and using SET with the POINT option.

Each of these topics will be covered in the following sections.

1. EACH SET STATEMENT ACTIVATES ONE READ POINTER.

Problem: An employee master file with all current employees must be "updated" by a transaction data set of new employees. A program must be written to add the new employees to the master file. The employee ID,
which is the observation number, is used by other applications as a pointer to access records directly. Therefore, when an employee leaves the company, that employee ID needs to be used again. At first glance, this does not seem to be a difficult programming problem.

Solution 1: FIGURE 1.1 contains the two SAS data sets, MASTER and NEWEMPS. FIGURE 1.2 is the first attempt to find the correct solution. The MASTER data set is read and if the employee name is missing, the NEWEMPS data set is read. When the end of the MASTER data set is reached, the rest of the NEWEMPS data set will be read until it is also empty. FIGURE 1.3, the output from the code in FIGURE 1.2, clearly demonstrates that this is not the correct solution. The point to be understood is that each SET statement activates one read pointer. Notice that there are two SET statements reading the NEWEMPS data set. Therefore, there are two read pointers for that data set and each one will begin at the first observation (by default) and read sequentially through the data set. When we reach the end of the MASTER data set, the second SET statement for the NEWEMPS data set is executed and the first observation is read. Thus TAYLOR becomes not only employee number two but also employee number six.

Solution 2: We can solve this problem by changing the program to contain only one physical SET statement to execute, activating only one read pointer, thereby reading the NEWEMPS data set sequentially. As seen in FIGURE 1.4 this goal can be achieved through the use of a LINK - RETURN. Only one physical SET statement appears in the code, only one SET statement is executed and the NEWEMPS data set is read once from the first observation to the last, as desired. The correct output can be seen in FIGURE 1.5. Other details of the correct code should also be noted. The two read operations
DATA NEWMASTR;
DO WHILE ( NOT LASTREC );
SET MASTER END = LASTREC;
IF EMPNAME = ' ' AND
NOT LASTNEW THEN
LINK GETNEW;
OUTPUT;
END;
IF LASTREC AND NOT LASTNEW THEN DO;
DO WHILE ( NOT LASTNEW );
EMPNO + 1;
LINK GETNEW;
OUTPUT;
END;
END;
STOP;
GETNEW:
SET NEWEMPS END = LASTNEW;
RETURN;
RUN;

FIGURE 1.4

in the data step are inside DO LOOPS; therefore
the WARNING message 'Program stopped due to
looping' would appear in the program log. To
prevent this WARNING message, an explicit
STOP is coded which will set the EDSF to 'Y' and
stop the DATA step. Without the STOP statement,

DATA NEWMASTR;
DO WHILE ( NOT LASTREC );
SET MASTER END = LASTREC;
IF EMPNAME = ' ' AND
NOT LASTNEW THEN
LINK GETNEW;
OUTPUT;
END;
IF LASTREC AND NOT LASTNEW THEN DO;
DO WHILE ( NOT LASTNEW );
EMPNO + 1;
LINK GETNEW;
OUTPUT;
END;
END;
STOP;
GETNEW:
SET NEWEMPS END = LASTNEW;
RETURN;
RUN;

FIGURE 1.4

in the data step are inside DO LOOPS; therefore
the WARNING message 'Program stopped due to
looping' would appear in the program log. To
prevent this WARNING message, an explicit
STOP is coded which will set the EDSF to 'Y' and
stop the DATA step. Without the STOP statement,

2. LOOK AHEAD MERGE WITH
FIRSTOBS OPTION.

Problem: The owner of a small florist shop
would like to see a report showing the
difference in income from month to month. The
report should show the difference in income
between February and January and then the
difference between March and February, etc.
The input data can be seen in FIGURE 2.1.

<table>
<thead>
<tr>
<th>MONTH</th>
<th>TOTAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAN</td>
<td>5000</td>
</tr>
<tr>
<td>FEB</td>
<td>17000</td>
</tr>
<tr>
<td>MAR</td>
<td>9000</td>
</tr>
<tr>
<td>APR</td>
<td>13000</td>
</tr>
<tr>
<td>MAY</td>
<td>20000</td>
</tr>
<tr>
<td>JUNE</td>
<td>19000</td>
</tr>
<tr>
<td>JUL</td>
<td>8000</td>
</tr>
<tr>
<td>AUG</td>
<td>8000</td>
</tr>
<tr>
<td>SEP</td>
<td>11000</td>
</tr>
<tr>
<td>OCT</td>
<td>16000</td>
</tr>
<tr>
<td>NOV</td>
<td>12000</td>
</tr>
<tr>
<td>DEC</td>
<td>14000</td>
</tr>
</tbody>
</table>

FIGURE 2.1

In the input data, the record for January is read
first, then the record for February is read,
followed by the record for March, and so on.

Solution: The desired code in FIGURE 2.2
shows that it is necessary to read the first
observation from the data set (the January
record) and "combine" that data with the data
from the second observation (February) in the

FIGURE 1.5

NEWMASTR DATA SET

<table>
<thead>
<tr>
<th>EMPNO</th>
<th>EMPNAME</th>
<th>DEPT</th>
<th>GROSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>JONES</td>
<td>901</td>
<td>95000</td>
</tr>
<tr>
<td>2</td>
<td>TAYLOR</td>
<td>903</td>
<td>32600</td>
</tr>
<tr>
<td>3</td>
<td>SMITH</td>
<td>902</td>
<td>47500</td>
</tr>
<tr>
<td>4</td>
<td>JOHNSON</td>
<td>904</td>
<td>41900</td>
</tr>
<tr>
<td>5</td>
<td>BROWN</td>
<td>901</td>
<td>17275</td>
</tr>
<tr>
<td>6</td>
<td>GREEN</td>
<td>901</td>
<td>39200</td>
</tr>
<tr>
<td>7</td>
<td>HARRIS</td>
<td>902</td>
<td>21800</td>
</tr>
</tbody>
</table>

FIGURE 1.5
DATA COMPARE (KEEP = MONTH DIFFER);
SET SALES (DROP = MONTH
(RENAME =
(TOTAL = LASTTOT)));
SET SALES (FIRSTOBS = 2);
DIFFER = TOTAL - LASTTOT;
RUN;

FIGURE 2.2

same execution of the data step. Specifically, on
execution one of the data step read observations
one and two; on the second execution read
observations two and three; etc. This is
accomplished by using the FIRSTOBS option on
the SET statement. This option tells the SAS
supervisor where to begin reading the data set.
Note that the variables in one of the SET
statements must be RENAMED to prevent the
values from being overwritten in the Program
Data Vector. The final report can be seen in

FIGURE 2.3

FIGURE 2.3. Again there are two read pointers
activated to the same data set and each will read
the same data set independently. The read
pointer associated with the second SET
statement will reach the end of the data set first
because the FIRSTOBS option forced the read
pointer to start with the second observation.
Therefore, the End of Data Step Flag will be set
to 'Y' when the second SET statement fails the
read operation. Also, note that the two SET
statements act like a one to one merge and
combine the two records into one observation in
the new data set. With the merge, however, the
read operation won't fail until both data sets are
empty.

3. resetting of In = variables.

Problem: A national department store chain
keeps the information about the products sold in
each department in separate SAS data sets. A
report is to be generated that combines all the
data sets to show the products carried in each
department during each month. FIGURE 3.1
shows some of the input data from two
departments. FIGURE 3.2 demonstrates the
desired report.

FIGURE 3.1

FIGURE 3.2
DATA NOTOK;
  MERGE GARDEN CLOTHING (RENAME = (PRODUCT = CPRODUCT));
  BY MONTH;
RUN;
PROC PRINT DATA = NOTOK;
  BY MONTH;
  ID MONTH;
  TITLE 'DATA NOTOK';
RUN;

FIGURE 3.3

Solution 1: FIGURE 3.3 shows the first attempt. As can be seen in FIGURE 3.4, the report could be improved upon by eliminating the "carry down" or retained value of PRODUCT or CPRODUCT. It should be noted that this "carry down" is what makes a one to many merge work, but is not desired in this particular many to many merge application. It is necessary to set either PRODUCT or CPRODUCT to missing when the input data are retained rather than read from the respective data set.

Solution 2: The above objective can be accomplished by using IN = variables for both the input data sets and setting the IN = variables, unconditionally, to zero for every execution of the data step. FIGURE 3.5 shows this correct solution. The IN = variables are initialized to 0 at the beginning of the data step. When an input record is read from the associated data set, the IN = variable is set to 1 and its value is retained until the next BY group. The important point to remember is that because the input data is read with a MERGE statement the IN = variable is not reinitialized to 0 until the BY group changes. The programmer must take control of the IN = variables by resetting them to 0 for every record, to determine if an observation from the input data set was read or retained. Then, the IN = variables only have a value of 1 when an input record is actually read from the associated data set. The IN = variables are available in the Program Data Vector and thus are available for the programmer to use, although they are not output to the new data set. The code in FIGURE 3.5 generates the output seen in FIGURE 3.2.

DATA REPORTOK;
  MERGE GARDEN (IN = INGARD)
     CLOTHING (IN = INCLOTH
     (RENAME = (PRODUCT = CPRODUCT));
  BY MONTH;
  IF INGARD = 0 THEN PRODUCT = ' ';
  IF INCLOTH = 0 THEN CPRODUCT = ' ';
  INGARD = 0;
  INCLOTH = 0;
RUN;
PROC PRINT DATA = REPORTOK;
  BY MONTH;
  ID MONTH;
  TITLE 'DATA REPORTOK';
RUN;

FIGURE 3.5
4. **POINT OPTION ON A SET STATEMENT TO READ A SPECIFIC OBSERVATION.**

**Problem:** To aid in budget planning for the upcoming year, a report must be generated that shows the percentage of the annual budget that was spent for each month during the previous year. FIGURE 4.1 shows the input data set with the MONTH, AMOUNT and a cumulative, year-to-date total called CUM. To compute the annual percentage, the value of CUM from the last record (the annual total) must be available for every execution of the data step.

<table>
<thead>
<tr>
<th>EXPENSE DATA SET</th>
</tr>
</thead>
<tbody>
<tr>
<td>MONTH</td>
</tr>
<tr>
<td>JAN</td>
</tr>
<tr>
<td>JAN</td>
</tr>
<tr>
<td>JAN</td>
</tr>
<tr>
<td>FEB</td>
</tr>
<tr>
<td>FEB</td>
</tr>
<tr>
<td>FEB</td>
</tr>
<tr>
<td>FEB</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>DEC</td>
</tr>
</tbody>
</table>

FIGURE 4.1

**Solution:** The code in FIGURE 4.2 demonstrates how this can be achieved. On the first execution of the data step only, a SET statement is executed using the POINT and NOBS option to read the LAST record. Remember, the value of NOBS is set in the Program Data Vector at compile time. Thus, when this SET statement executes, the number of observations in the data set is available in the PDV as the value of NUM_OBS and the POINT option can be used to point specifically to the last record. Notice also that the variable CUM is RENAMED to ANNUAL. Therefore, the variable ANNUAL is present in the Program Data Vector and is never reinitialized to missing because it was read with a SET statement. Thus, ANNUAL is made available for every record and the monthly percentage can easily be calculated. Note also that the POINT option on the first SET statement precludes the use of a BY statement, therefore FIRST or LAST processing must be simulated with the LAG function. FIGURE 4.3 shows the final report and the desired results.

```sas
DATA MONTHLY;
(KEEP = EXPMONTH PERCENT);
IF _N_ = 1 THEN SET EXPENSES (KEEP = CUM (RENAME = (CUM = ANNUAL)));
POINT = NUM_OBS;
NOBS = NUM_OBS;
SET EXPENSES (KEEP = MONTH AMOUNT);
END = LASTREC;
/* force lag to execute */
/* for every record */
LASTMON = LAG (MONTH);
IF MONTH ^= LASTMON AND _N_ > 1 THEN DO; /* new month */
EXPMONTH = LASTMON;
PERCENT = (MONTHTOT / ANNUAL) * 100;
OUTPUT MONTHLY;
MONTHTOT = 0; /* reinit */
END;
MONTHTOT + AMOUNT;
IF LASTREC THEN DO; /* process last month */
EXPMONTH = MONTH;
PERCENT = (MONTHTOT / ANNUAL) * 100;
OUTPUT MONTHLY;
END; /* process last month */
RUN;
```

FIGURE 4.2
MONTHLY DATA SET

<table>
<thead>
<tr>
<th>EXPMONTH</th>
<th>PERCENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>JAN</td>
<td>3.55%</td>
</tr>
<tr>
<td>FEB</td>
<td>15.08%</td>
</tr>
<tr>
<td>MAR</td>
<td>5.92%</td>
</tr>
<tr>
<td>APR</td>
<td>8.55%</td>
</tr>
<tr>
<td>MAY</td>
<td>13.15%</td>
</tr>
<tr>
<td>JUNE</td>
<td>12.50%</td>
</tr>
<tr>
<td>JUL</td>
<td>5.26%</td>
</tr>
<tr>
<td>AUG</td>
<td>5.26%</td>
</tr>
<tr>
<td>SEP</td>
<td>7.24%</td>
</tr>
<tr>
<td>OCT</td>
<td>6.49%</td>
</tr>
<tr>
<td>NOV</td>
<td>7.89%</td>
</tr>
<tr>
<td>DEC</td>
<td>9.21%</td>
</tr>
</tbody>
</table>

5. USE OF THE NOBS OPTION AT COMPILE TIME.

**Problem:** Every evening a program is run to update a master file. There is extensive editing code to verify that the records to be modified, added or deleted are valid. Following the edit data step there is a complex update step that generates several output data sets and update reports. If there are no good records to process following the edit data step it would be reasonable to forego the update data step.

**Solution:** This can be accomplished by using the NOBS variable which is added to the PDV and initialized at COMPILE time. The macro in FIGURE 5.1 demonstrates this concept. The focus of this example is the DATA _NULL_ step which is going to create a macro variable (&N_OBS) with its value set to the number of observations in the GOODRECS data set. At COMPILE time the value of the data step variable, N_OBS, is initialized in the Program Data Vector. At EXECUTION time the SET statement will never execute because 'IF 0' is never true but the program still creates the macro variable &N_OBS with the SYMPUT function.

Again, the STOP statement is explicitly coded to prevent the WARNING message 'Program stopped due to looping'. To take advantage of the power of %IF logic this code must be within a macro. Note, the POINT option on the SET statement is used in this mainframe environment because the NOBS option is only available with the POINT option. On the PC or a mini computer the NOBS option does not require the use of POINT.

```
%MACRO UPDAILY;
/* this macro executes */
/* the update every day */
DATA GOODRECS;
 (edit code)
RUN;
/* records to process?? */
DATA _NULL_;
IF 0 THEN SET GOODRECS
POINT = _N_ NOBS = N_OBS ;
CALL SYMPUT( 'N_OBS',
PUT ( N_OBS , 5.));
STOP;
RUN;
%IF &N_OBS > 0 %THEN
%DO; /* exec if records */
DATA MASTER BADRECS;
 (update code)
RUN;
%END;
%MEND UPDAILY;
```

6. JOIN -- ALL COMBINATIONS MERGE.

**Problem:** A company has a benefit program that provides employees with improved benefits as the employee is promoted. As can be seen in FIGURE 6.1, an employee in CATEGORY 1 has four potential benefits while an employee in CATEGORY 5 has only two potential benefits. In the EMPLOYEE data set, FIGURE 6.2, each employee has a CATEGORY associated with their position. It is necessary to produce a report that
### EMPLOYEE DATA SET

<table>
<thead>
<tr>
<th>EMPNAME</th>
<th>CATEGORY</th>
<th>HOURLY</th>
</tr>
</thead>
<tbody>
<tr>
<td>MANN</td>
<td>1</td>
<td>25.00</td>
</tr>
<tr>
<td>SMITHSON</td>
<td>2</td>
<td>23.00</td>
</tr>
<tr>
<td>JAMES</td>
<td>2</td>
<td>21.00</td>
</tr>
<tr>
<td>HOUSEMAN</td>
<td>3</td>
<td>20.00</td>
</tr>
<tr>
<td>GARRETT</td>
<td>3</td>
<td>19.00</td>
</tr>
<tr>
<td>ROBBINS</td>
<td>4</td>
<td>20.00</td>
</tr>
<tr>
<td>STILES</td>
<td>4</td>
<td>18.00</td>
</tr>
<tr>
<td>HOWELL</td>
<td>4</td>
<td>16.00</td>
</tr>
<tr>
<td>DUBOIS</td>
<td>4</td>
<td>15.00</td>
</tr>
<tr>
<td>JENKINS</td>
<td>5</td>
<td>12.00</td>
</tr>
<tr>
<td>PAYNE</td>
<td>5</td>
<td>10.00</td>
</tr>
</tbody>
</table>

**FIGURE 6.1**

### BENEFITS DATA SET

<table>
<thead>
<tr>
<th>CATEGORY</th>
<th>BENEFIT</th>
<th>VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DENTAL</td>
<td>200.00</td>
</tr>
<tr>
<td>1</td>
<td>FULL MEDICAL</td>
<td>1000.00</td>
</tr>
<tr>
<td>1</td>
<td>BONUS</td>
<td>2000.00</td>
</tr>
<tr>
<td>1</td>
<td>4 WKS VACA</td>
<td>160</td>
</tr>
<tr>
<td>2</td>
<td>FULL MEDICAL</td>
<td>1000.00</td>
</tr>
<tr>
<td>2</td>
<td>BONUS</td>
<td>1500.00</td>
</tr>
<tr>
<td>2</td>
<td>3 WKS VACA</td>
<td>120</td>
</tr>
<tr>
<td>3</td>
<td>FULL MEDICAL</td>
<td>1000.00</td>
</tr>
<tr>
<td>3</td>
<td>3 WKS VACA</td>
<td>120</td>
</tr>
<tr>
<td>4</td>
<td>HALF MEDICAL</td>
<td>500.00</td>
</tr>
<tr>
<td>4</td>
<td>3 WKS VACA</td>
<td>120</td>
</tr>
<tr>
<td>5</td>
<td>HALF MEDICAL</td>
<td>500.00</td>
</tr>
<tr>
<td>5</td>
<td>2 WKS VACA</td>
<td>80</td>
</tr>
</tbody>
</table>

**FIGURE 6.2**

will show each employee the value of their specific benefits for the previous year. A simple merge would work in this case if there were only one benefit or one employee for each category. If there are multiple benefits for any category or multiple EMPLOYEE records for any category, however, the merge would not combine the BENEFITS with the EMPLOYEES in the desired manner.

**Solution 1:** To produce the desired output, a JOIN or ALL COMBINATIONS of the BY group MERGE must be performed. This is accomplished by creating a pointer data set for each of the two data sets. These pointer data sets contain one record for each value of CATEGORY and two other variables whose values are the first and last observation number for the given CATEGORY in the original data sets. These pointer data sets will be used to read an employee record and combine it with each of the records in the CATEGORY from the BENEFITS data set, creating one record for each benefit for each employee. **FIGURES 6.3 and 6.4**
demonstrate the creation of the pointer data sets. The code for the two data sets is almost identical with the exception of the names used for the start and stop variables. FIGURE 6.5 shows the resulting output pointer data sets.

<table>
<thead>
<tr>
<th>PNTREMP DATA SET</th>
<th>CATEGORY</th>
<th>STARTE</th>
<th>STOPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>6</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PNTRBEN DATA SET</th>
<th>CATEGORY</th>
<th>STARTB</th>
<th>STOPB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>8</td>
<td>9</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>11</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>13</td>
<td></td>
</tr>
</tbody>
</table>

FIGURE 6.5

The code in FIGURE 6.6 indicates one solution to the programming problem. The pointer data sets are merged together to make the pointers available in the data step. The first DO LOOP ranges from the start to the stop variable corresponding to all observations for the CATEGORY in the EMPLOYEE data set. Within that DO LOOP a second DO LOOP ranges from the start through the stop variable corresponding to all observations for the CATEGORY in the BENEFITS data set. For each employee, all the BENEFIT records for the CATEGORY are read creating one record for every EMPLOYEE-BENEFIT combination. The output seen in FIGURE 6.7 shows correct results from this example. Depending on the specific application, this code could work perfectly well. Note, however that the MERGE statement assumes a one to one correspondence between the two pointer data sets. For every value of CATEGORY in the BENEFITS data set there is at least one employee and for every employee CATEGORY there is at least one BENEFIT. If one or more values of the CATEGORY variable were not present in the EMPLOYEE or BENEFITS data set, this code would not produce the desired results.

<table>
<thead>
<tr>
<th>JOIN1 DATA SET</th>
<th>EMPNAME</th>
<th>BENEFIT</th>
<th>TOTVAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>MANN</td>
<td>DENTAL</td>
<td>200.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FULL MEDICAL</td>
<td>1000.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BONUS</td>
<td>2000.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 WKS VACA</td>
<td>4000.00</td>
<td></td>
</tr>
<tr>
<td>SMITHSON</td>
<td>FULL MEDICAL</td>
<td>1000.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BONUS</td>
<td>1500.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 WKS VACA</td>
<td>2760.00</td>
<td></td>
</tr>
<tr>
<td>JAMES</td>
<td>FULL MEDICAL</td>
<td>1000.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>BONUS</td>
<td>0.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 WKS VACA</td>
<td>2520.00</td>
<td></td>
</tr>
<tr>
<td>HOUSEMAN</td>
<td>FULL MEDICAL</td>
<td>1000.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 WKS VACA</td>
<td>2400.00</td>
<td></td>
</tr>
<tr>
<td>GARRETT</td>
<td>FULL MEDICAL</td>
<td>1000.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 WKS VACA</td>
<td>2280.00</td>
<td></td>
</tr>
<tr>
<td>ROBBINS</td>
<td>HALF MEDICAL</td>
<td>500.00</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 WKS VACA</td>
<td>2400.00</td>
<td></td>
</tr>
</tbody>
</table>

FIGURE 6.7

Solution 2: Suppose the EMPLOYEE data set had no employees with CATEGORY = 3. The pointer data sets would be created in the same way, but the values of STARTE and STOPE would
DATA NEWEMPS;
    MERGE EMPLOYEE (IN = INEMP)
             PNTRBEN;
    BY CATEGORY;
    IF INEMP; /* keep emp recs */
    RUN;
DATA JOIN2 (KEEP = EMPNAME
             BENEFIT TOTVAL);
SET NEWEMPS;
DO I = STARTB TO STOPB;
    SET BENEFITS POINT = I;
    (perform calculations)
    (to compute TOTVAL)
    OUTPUT;
END; /* do I = */
RUN;

FIGURE 6.8
differ from the originals. The two data steps in FIGURE 6.8 demonstrate the solution to this problem. Note, the EMPLOYEE data set (not the employee pointer data set) and the BENEFITS pointer data set are MERGED with a BY statement in a separate data step and only the records with input from the EMPLOYEE data set are kept. This merge adds the BENEFITS pointers to the data set for each employees’ CATEGORY. Set with the POINT in the next data step is used to read, for each employee, all the appropriate BENEFITS records. The use of SET with a POINT in the second data step prohibits the two data steps from being combined into one.

7. CONCLUSION

The six examples presented in this paper were designed to demonstrate advanced techniques for the use of the SET and MERGE statements. In addition, the code presented reinforces some of the concepts presented in reference 1. It has demonstrated the power available to the SAS programmer by efficient use of many of the options available with the SET and MERGE statements. By understanding how the SAS supervisor affects the Program Data Vector, the SAS programmer can more easily solve programming problems.

* SAS is a registered trademark of SAS Institute, Inc.

References:


ACKNOWLEDGEMENTS:

The input and assistance of the following people was greatly appreciated: Katie Hubbell, Ian Whitlock, Josh Sharlin, Judie Mopsik, and Don Henderson.

The authors can be contacted at:

SOFTWARE APPLICATIONS AND TRAINING DIVISION

ORI, INC.
SUITE 501
601 INDIANA AVENUE, N.W.
WASHINGTON, D.C. 20004

TELEPHONE: (202) 737-2666

1177