An Introduction to Reshaping (TRANSPOSE) and Combining (MATCH-MERGE) SAS® Data Sets

Mike Zdeb, University@Albany School of Public Health, Rensselaer, NY

ABSTRACT

SAS offers many ways to analyze and present data. However, since data may not be stored in a format required for a particular analysis and/or mode of presentation, another set of SAS tools that reorganize data must be used first --- those that reshape (turn variables into observations and vice-versa) and those that combine data sets. Just as one can select among a number of ways to analyze and present data, reshaping and combining of data sets may be done in several different ways. This paper focuses on the use of PROC TRANSPOSE to reshape data sets and data step match-merge to combine data sets. The effects of using various PROC TRANSPOSE options and statements are illustrated as are various types of match-merge situations (one-to-one, one-to-many, many-to-many) and how to control the results of match-merge using the IN= data set option. Alternative methods are briefly discussed: using arrays to reshape data; using PROC SQL to combine data sets.

INTRODUCTION

One of the many strengths of SAS software is the number of tools is offers for data organization, or perhaps reorganization s a better term. Prior to using any of the SAS procedures for data analysis or the ODS tools for data presentation, it is not uncommon to spend a lot of time and effort putting data in a form that make it amenable to analysis and presentation. Sometimes, if you want to conduct an analysis across observations in a data set, you can use SAS procedures. Or, if you want to conduct an analysis within observations, you can use SAS functions. However, there are occasions when neither a procedure nor a function will suffice. Your data may be arranged in such a way that the only way to complete a given task is to first rearrange the data set. Rearranging in this context means converting variables into observations or observations into variables. Both types of rearranging can be accomplished with PROC TRANSPOSE.

Another common situation is to have data in multiple data sets. Analysis and/or presentation of the data might require that all the data sets be combined. Combination of two or more data sets might mean simply concatenating the observations by means of a SET statement or PROC APPEND. However, sometimes it involves combining the information in observations with information in observations in another data set by using a MERGE statement rather than SET.

The following examples illustrate first how to rearrange data sets and then how to combine data sets. Rearranging is accomplished with PROC TRANSPOSE while combining is done with data step match-merges.

REARRANGING DATA SETS

Problem #1...you have SAS data set with observations that contain up to five occurrences of a 3-digit medical diagnosis. Each observation represents one person and your task is to create a table that shows how often each diagnosis occurs within the data set. The following data step creates the data set to be used in the examples.

```
* EXAMPLE 1;
data many_dx;
infile datalines missover;
input id : $2. (dx1-dx5)(:$3.);
datalines;
01 647 641 650 428
02 428 416
03 642 674 648
04 641 416 648 647 641
;
run;
```

One approach is to use PROC FREQ on each of the five diagnoses and add the results of the five tables...

```
proc freq data=many_dx;
tables dx1-dx5;
run;
```

Another approach is to rearrange the data creating a new data set that contains only one variable, DIAG, with one observation for each diagnosis in the original data set. You create one observation for each diagnosis in the original data set.

```
* EXAMPLE 2;
data all_dx;
set many_dx;
diag = dx1; if diag ne ' ' then output; ❶
diag = dx2; if diag ne ' ' then output;
diag = dx3; if diag ne ' ' then output;
diag = dx4; if diag ne ' ' then output;
diag = dx5; if diag ne ' ' then output;
keep diag; ❷
run;
```

A new variable DIAG is added to the data set. Every time an observation is read with a SET statement, the value of each of the five diagnoses (DX1 through DX5) is placed in the variable DIAG. If the value is non-missing, an observation is added to the data set. Each observation in the original data set can contribute up to five observations to the new data set. Only the variable diag is kept in the new data set.

```
Obs    diag
  1    647
  2    641
  3    650
  4    428
  5    428
  6    416
  7    642
  8    674
  9    648
 10    641
 11    416
 12    648
 13    647
 14    641
```

You can then use PROC FREQ to create a table of one variable DIAG that represents all the diagnoses in the original data set.

```
proc freq data=all_dx;
table diag;
run;
```

The above data step turned variables into observations. That is one of the functions of PROC TRANSPOSE. Rather than use a data step, use PROC TRANSPOSE to rearrange the data.

```
* EXAMPLE 3;
proc transpose data=many_dx out=all_dx; ❶
var dx1-dx5; ❷
run;
```

The data set to be rearranged is specified after DATA= while the data set to be created is specified after OUT= ❶. If you do not specify a name for the new data set, SAS will provide a name for the data set with the form DATA1, DATA2, etc. The variables to be transposed are specified in a VAR statement ❷. If no VAR statement is used, PROC TRANSPOSE will use all the numeric variables in the DATA= data set. In this example, since all the variables are character, without a VAR statement, PROC TRANSPOSE will create a new data set with zero observations and five variables (one variable for each observation in the DATA= data set. Since there was a VAR statement, PROC TRANSPOSE produced the following...

```
Obs    _NAME_    COL1    COL2    COL3    COL4
 1      dx1      647     428     642     641
 2      dx2      641     416     674     416
 3      dx3      650             648     648
 4      dx4      428                     647
 5      dx5                              641
```

PROC TRANSPOSE turned the variables DX1through DX5 into observations. However, rather than producing a data set with only one variable as was done in example 2, there are five variables. There is a variable _NAME_ that contains the names of the transposed variables followed by one new variable with the prefix COL for each observation in the original data set. If there had been 1,000 observations in data set MANY_DX, data set ALL_DX would have 1,001 variables with the names _NAME_ and COL1 through COL1000. The prefix COL is the default prefix assigned by PROC TRANSPOSE

There may be occasions when the data set that results from example 3 is exactly what you want. That is not the case here since we want a data set with only one variable that contains all the values of the diagnoses. That can be accomplished by adding another statement to PROC TRANSPOSE.

```
* EXAMPLE 4;
proc transpose data=many_dx out=all_dx;
var dx1-dx5;
by id; ❶
run;
```

A BY statement is added, specifying ID as the by-variable ❶. As with other instances of using by-variables in SAS, the DATA= data set must be sorted (or indexed) in ascending order of the by-variable (note: there is one instance in SAS where this is not true...do you know when that occurs?). The new data set looks as follows...

```
Obs    id    _NAME_    COL1
  1    01     dx1      647
  2    01     dx2      641
  3    01     dx3      650
  4    01     dx4      428
  5    01     dx5
  6    02     dx1      428
  7    02     dx2      416
  8    02     dx3
  9    02     dx4
 10    02     dx5
 11    03     dx1      642
 12    03     dx2      674
 13    03     dx3      648
 14    03     dx4
 15    03     dx5
 16    04     dx1      641
 17    04     dx2      416
 18    04     dx3      648
 19    04     dx4      647
 20    04     dx5      641
```

Variables are converted to observations within by-groups. Since there is only one observation in each by-group, there are now five observations per observation in the original data and only one variable with the prefix COL. This new data set is very close to the target data set. As with the results from example 3, this may be the desired output. But, a few data set options can be added to PROC TRANSPOSE and the data set will look identical to that produced by the data step in example 4.

```
* EXAMPLE 5;
proc transpose data=many_dx
               out=all_dx (keep=col1 ❶ rename=(col1=diag) ❷ where=(diag ne '') ❸);
var dx1-dx5;
by id;
run;
```

The only variable we want in the new data set is COL1 ❶. That variable is renamed to DIAG ❷. Only observations with non-missing values for DIAG are output to the new data set ❸. The order of the KEEP, RENAME, and WHERE data set options does not matter. Regardless on the order, SAS processes the KEEP option first so you must keep the variable COL1(the default name assigned by PROC TRANSPOSE). The RENAME option is processed prior to the WHERE option so you must use the new variable name in the WHERE option.

```
Obs    diag
  1    647
  2    641
  3    650
  4    428
  5    428
  6    416
  7    642
  8    674
  9    648
 10    641
 11    416
 12    648
 13    647
 14    641
```

The data set produced by PROC TRANSPOSE in example 5 is now identical to that produced with a data step in example 2. There are a few other options available in PROC TRANSPOSE, but in the situation where you want to create a new data set with observations based on variables in the original data set, the above examples show most of the options (both PROC and data set) that are important.

The data step in example 1 could be rewritten creating a data set with all numeric rather than character variables.

```
* EXAMPLE 6;
data many_dx;
infile datalines missover;
input id dx1-dx5;
datalines;
01 647 641 650 428
02 428 416
03 642 674 648
04 641 416 648 647 641
;
run;
```

As stated earlier, the default behavior of PROC TRANSPOSE is to use all numeric variables in a data set. If you use PROC TRANSPOSE with no VAR statement and the data set from example 6...

```
* EXAMPLE 7;
proc transpose data=many_dx out=all_dx;
run;
```

the new data set is similar to that produced in example 3, except that there is an additional observation based on the values of the variable ID in the original data. Also, the variables COL1-COL4 in the new data set are numeric rather than character as they were in example 3.

```
Obs    _NAME_    COL1    COL2    COL3    COL4
 1      id          1       2       3       4
 2      dx1       647     428     642     641
 3      dx2       641     416     674     416
 4      dx3       650       .     648     648
 5      dx4       428       .       .     647
 6      dx5         .       .       .     641
```

Problem #2...you have a SAS data set with observations that contain an account number, a month (in the form of 1 for January, 2 for February, etc.), and a dollar value indicating the amount deposit in money market account. You would like to create a report showing the monthly deposits for each person in your data set (as indicated by the account number). You think that it would be easier to create the report if the data set had one observation per person with all the deposits in that observation. The following data step creates the data set used in the examples.

```
* EXAMPLE 8;
data deposits;
input account : $2. month deposit @@;
datalines;
01 1 100 01 4 50  01 6 200
02 2 50  02 3 100
03 1 50  03 2 50  03 3 50 03 4 50  03 5 50  03 6 50
;
run;
```

As was done in the last problem, you could use a data step to rearrange the data. However, the data step to put all the data for into a single observation per person is mor complex than in the last problem, requiring the use of FIRST. and LAST. variables. PROC TRANSPOSE will be used without even showing such a data step.

```
* EXAMPLE 9;
proc transpose data=deposits out=acct_deposits; ❶
run;
```

PROC TRANSPOSE is used without any extra statements ❶. Only the input and output data set names are specified. Without a VAR statement, the two numeric variables in the original data set (MONTH, DEPOSIT) are transposed.

```
Obs    _NAME_     COL1    COL2    COL3    COL4    COL5    COL6    COL7    COL8    COL9    COL10    COL11
 1     month         1       4       6       2       3       1       2       3       4        5        6
 2     deposit     100      50     200      50     100      50      50      50      50       50       50
```

The new data set has one observation for each transposed variable and twelve variables, one for each observation in the original data plus the variable _NAME_. The data are rearranged, but not in the form of one observation per person. That can be accomplished by adding another statement to PROC TRANSPOSE.

```
* EXAMPLE 10;
proc transpose data=deposits out=acct_deposits;
by account;
run;
```

A BY statement rearranges the data one person at a time rather than rearranging all the data at one time. The DATA= data set must be sorted (or indexed) in ascending order by the variable ACCOUNT. Adding the BY statement results in a data set that has two observations per person since there are two numeric variables in the original data.

```
Obs     account     _NAME_      COL1     COL2     COL3     COL4     COL5     COL6
 1         01       month          1        4        6        .        .        .
 2         01       deposit      100       50      200        .        .        .
 3         02       month          2        3        .        .        .        .
 4         02       deposit       50      100        .        .        .        .
 5         03       month          1        2        3        4        5        6
 6         03       deposit       50       50       50       50       50       50
```

A few more changes are needed.

```
* EXAMPLE 11;
proc transpose data=deposits out=acct_deposits (drop=_name_ ❶) prefix=dep; ❷
var deposit; ❸
by account;
run;
```

The variable _NAME_ is dropped from the OUT= data set ❶. The default variable names COL1 through COL6 are replaced with DEP1 through DEP6 through use of a PREFIX option ❷. Notice that the text following PREFIX= is not in quotes even though it is not the name of a variable. Rather, it is text meant to replace the prefix COL in the names of the variables in the new data set. Adding the VAR statement limits the transposed variables to DEPOSIT ❸.

```
Obs     account     dep1     dep2     dep3     dep4     dep5     dep6
 1         01        100       50      200        .        .        .
 2         02         50      100        .        .        .        .
 3         03         50       50       50       50       50       50
```

Notice that you did not have to tell PROC TRANSPOSE how many new variables (DEP1-DEP6) are needed to accommodate the largest number of deposits for a particular person. As stated earlier in this problem, you could have used a data step. If you did, you would have had to have know that six new variables were needed prior to writing that data step. Notice that within observations 1 and 2, the values of DEPOSIT in the original data set are used to fill the values of DEP1 through DEP6 starting with DEP1. In the original data set, there was a variable MONTH than indicated the month of deposit. The deposits for the first person were made in months 1, 4, and 6. For the second person, deposits were made in months 2 and 3. If you want the numeric suffix on the variables DEP1-DEP6 to indicate the month of deposit, you can use a ID statement.

```
* EXAMPLE 12;
proc transpose data=deposits out=acct_deposits (drop=_name_) prefix=dep;
var deposit;
by account;
id month; ➊
run;
```

The variable MONTH is used as an ID variable ➊ telling PROC TRANSPOSE to create the suffix on the variable names DEP1-DEP6 based on the value of the variable MONTH. Note, you can think of this task as filling a spreadsheet using the data from data set DEPOSITS: the ID statement uses a variable that controls what column is filled; the BY statement uses a variable what row is filled; the VAR statement specifies the variable whose value is used to fill the cell in the spreadsheet.

```
Obs    account    dep1    dep2    dep3    dep4    dep5    dep6
 1       01        100      .       .      50       .      200
 2       02          .     50     100       .       .        .
 3       03         50     50      50      50      50       50
```

What if the original data had text strings for the values of month rather than numbers? Month would then have to be a character variable.

```
* EXAMPLE 13;
data deposits;
input account : $2. month : $3. deposit @@;
datalines;
01 JAN 100 01 APR 50  01 JUN 200
02 FEB 50  02 MAR 100
03 JAN 50  03 FEB 50  03 MAR 50 03 APR 50  03 MAY 50  03 JUN 50
;
run;
```

When an ID variable is specified, PROC TRANSPOSE tries to use the value of that variable to create the names of variables in the new data set. If the values are numeric (as in the data set created in example 8), PROC TRANSPOSE uses an underscore as the default prefix. If no PREFIX option had been use in example 12, the names of the new variables would have been _1 through _6, not DEP1 through DEP6. When the values of the ID variable are text strings , PROC TRANSPOSE uses that text to name the new variables. If the text contains characters not allowed in SAS variable names, those characters are replaced with underscores.

```
* EXAMPLE 14;
proc transpose data=deposits out=acct_deposits (drop=_name_); ➊
var deposit;
by account;
id month; ➋
run;
```

No PREFIX option is used. Since the values of the variable MONTH are the first three characters of the names of the months, PROC TRANSPOSE uses those values to create the names of the variables in the new data set.

```
Obs    account    JAN    FEB    MAR    APR    MAY    JUN
 1       01       100      .      .     50      .    200
 2       02         .     50    100      .      .      .
 3       03        50     50     50     50     50     50
```

Other PROC TRANSPOSE Options and Statements

There are several other PROC TRANSPOSE options and statements that are not covered in the examples. The options are LET, LABEL, and NAME. The statements are COPY and IDLABEL. You can read about these options and statements in the SAS on-line help. The most important and frequently used options are covered in the examples.

Rearranging Data Sets with Arrays

Example 2 showed a data step alternative to PROC TRANSPOSE for rearranging data. There is another version of a data step alternative and it involves the use of an array.

```
* EXAMPLE 15;
data all_dx;
set many_dx;
array dx(5); ❶
do j=1 to 5; ❷
   diag=dx(j); ❸
   if diag ne ' ' then output; ❹
end;
keep diag;
run;
```

An ARRAY statement creates an array name DX (based on the values DX1-DX5 in the data set MANY_DX) ❶. A DO loop is used to look at the five diagnoses within each observation ❷. The variable DIAG is created based on the value of DX1 through DX5 ❸. If DIAG is non-missing, an observation is written to data set ALL_DX ❹.

Examples 9 through 12 showed how to use PROC TRANSPOSE to rearrange a data set that had one observation for variouis combinations of account number and month. The new data set had one observation per account. Once again, you can do the same rearranging uisng an array.

```
* EXAMPLE 16;
data acct_deposits;
retain dep1-dep6; ❶
set deposits;
by account; ❷
array dep(6); ❸
if first.account then do j=1 to 6; ❹
   dep(j) = .;
end;
dep(month) = deposit; ❺
if last.account then output; ❻
keep account dep1-dep6;
run;
```

A RETAIN statement is necessary since observations are built as the data step reads observations within a given account number ❶. No observation is written to the new data set until after the last observation within an account number is read. Without the RETAIN statement, values of DEP1-DEP6 that are created for each account number would be set to missing in each cycle of the data step. The DEPOSITS data set is read by account, creating FIRST.ACCOUNT and LAST.ACCOUNT variables for use within the data step ❷. The array DEP is created to hold the values of the variable DEPOSIT in the original data set ❸. The values of DEP1-DEP6 are initialized as missing at the start of each by-group ❹. The value of the variable MONTH is used to place the value of the variable DEPOSIT in the appropriate DEP variable (1 through 6) ❺. After the last observation within a by-group has been read, an observation is written to the new data set ❻. As opposed to when PROC TRANSPOSE was used to rearrange the same data as used in example 16, you must know how many new variables to create prior to writing the data step, in this case DEP1-DEP6. PROC TRANSPOSE makes that determination for you.

The data step plus an array approach to rearranging data used in examples 15 and 16 does have some advantage over PROC TRANSPOSE in that it is more flexible. If the data used in example 16 had contained both a DEPOSIT and WITHDRAWAL and you wanted to create a new data set with DEP1-DEP6 and WDR1-WRD6, you could not do that with a single use of PROC TRANSPOSE. However, that would be an easy task with one data step plus arrays. If the data used in example 15 contained both DX1-DX5 and PR1-PR5 (5 diagnoses and 5 procedures) in each observation and you wanted to create a new data set with only two variables, DIAG and PRO, you could not do that with a single use of PROC TRANSPOSE. Again, that is an esay task for a single data step and arrays.

COMBINING DATA SETS

There are a number of ways to combine SAS data sets.

- concatenate - stack data sets with a SET statement (place one after another)
- interleave - stack data sets with a SET statement, but form the result In order by one or more variables present in the data sets through use of a BY statement
- PROC APPEND - stack data sets (place one after another)
  NOTE: both concatenation, interleaving, and PROC APPEND result in a data set that has as many observations as the sum of all the observations in the data sets being combined
- one-to-one merge - use a MERGE statement to combine observation one in data set one with observation one in data set two, observation two-with-two, three-with-three, etc.
  NOTE: the data set resulting from one-to-one merge will have as many observations as the number of observations in the largest data set involved in the merge
- matched-merge - use a MERGE statement and a BY statement to combine observations in data sets based upon the value of one or more variables
  NOTE: the number of observations in the resulting data set depends on the content of the data sets being combined and on the matching options being used
- UPDATE/MODIFY - change the values of the variables in certain observations in a data set based upon the values of variables in another data set (in case you could not answer the question posed after example 4, when a by-variable is used with a MODIFY statement, data sets need not be sorted or indexed according t values of the by-variable)
  NOTE: as with matched-merge, the number of observations in the resulting data set depends upon the content of the data sets being combined and on the matching options being used
- PROC SQL - either stack or combine observations

Only match-merge is discussed in the following examples, with a mention of PROC SQL as an alternative in some situations when a match-merge will not work properly

Problem #1...you have two SAS data sets, one with information collected on individuals in the month of January, the other with similar information collected in February. You want to create a new data set with one observation per person that contains all the information from both months. Each data set contains a weight for each person and once the data sets are combined, you want to compute the weight change from January to February. The following data step creates the data sets use in the examples.

```
* EXAMPLE 1;
data jan;
input ssn weight zip : $5.; ➊
format ssn ssn.; ➋
datalines;
001001234 180 12203 ➌
123456789 150 13502
888888888 200 14001
987654321 120 12345
;
run;

data feb;
input ssn weight;
datalines;
001001234 160 ➌
123456789 145
987654321 125
999999999 150
;
run;
```

There are a few differences between the two data sets: data set JAN contains an extra variable, ZIP ➊; the variable SSN is formatted in data set JAN ➋; there are three people common to both data sets and one unique to each data set. A match-merge requires that each data set contain a variable common to all data sets mentioned in the MERGE statement. In these data sets, that variable is SSN. The variable must have the same type in the data sets and should also have the same length, though you can match-merge data sets based on the values of variables with different lengths (not advised). A new data set is created with a match-merge. There is only one occurrence of the values of the by-variable SSN in each data set to be merged. This is know as a ***one-to-one*** match-merge.

```
* EXAMPLE 2;
data jan_feb;
merge jan feb; ❶
by ssn; ❷
run;
```

The two data sets are combined using a MERGE statement ❶. Use of a BY statement makes this a match-merge rather than a simple merge and forces observations to be combined based on the value of the by-variable SSN ❷. Data set JAN_FEB contains the following observations...

```
Obs        ssn        weight     zip
 1     001-00-1234      160     12203
 2     123-45-6789      145     13502
 3     888-88-8888      200     14001
 4     987-65-4321      125     12345
 5     999-99-9999      150
```

The default behavior of a match-merge is: the combined data set contains all matched observations, plus all unmatched observations from all data sets in the MERGE statement; the combined data set contains all variables common to all data sets in the MERGE Statement, plus all variables unique to each data set in the MERGE statement . You can see the common variables SSN and WEIGHT and the variable ZIP that unique to data set JAN. Also, you can see that observation #5 is unmatched since it has no value for ZIP (that only occurs in data set JAN), but it is hard to see that observation #3 is also unmatched since it has values for all variables. For the matched observations (#s 1, 2, and 4) the value of the variable WEIGHT is taken from data set FEB. The default behavior for a match-merge (or simple merge) is that when variables are common to merged data sets, the combined data set will have the value from the data set that occurs last in the MERGE statement. Also notice that though the variable SSN was formatted only in data set JAN, it is also formatted in the combined data set JAN_FEB.

The MERGENOBY option can be set to issue a warning or the stop a match-merge if you forget to se a BY statement. The default setting of MERGENOBY is NOWARN. The two other possible settings are WARN and ERROR.

```
* EXAMPLE 3;
options mergenoby=error; ❶

data jan_feb;
merge jan feb; ❷
run;
```

An options statement is used to stop a merge is no BY statement is present ❶. No BY statement is used for the merge of data sets JAN and FEB ❷. The LOG contains the following...

```
ERROR: No BY statement was specified for a MERGE statement.
NOTE: The SAS System stopped processing this step because of errors.
```

When merged data sets contain variables that have identical names, a RENAME data set option can be used to preserve the values of the variables in the merged data sets. Otherwise, matched observations contain the value of the variable in the dat set mentioned last in the merge statement (as shown in the results of example 2).

```
* EXAMPLE 4;
data jan_feb;
merge jan (rename=(weight=wt1)) feb (rename=(weight=wt2)); ❶
by ssn;
diff = wt2 - wt1; ❷
run;
```

The goal of merging the data sets was to compute the weight change from January to February. The variable WEIGHT is renamed using data set options in the MERGE statement ❶. The difference in weight is calculated.

```
Obs        ssn        wt1     zip     wt2    diff
 1     001-00-1234    180    12203    160     -20
 2     123-45-6789    150    13502    145      -5
 3     888-88-8888    200    14001      .       .
 4     987-65-4321    120    12345    125       5
 5     999-99-9999      .             150       .
```

It is now easy the matched versus unmatched observations. As stated previously, the default behavior of a match-merge is to create a data set with all matched and unmatched observations. This default behavior can be changed using IN= data set options.

```
* EXAMPLE 5;
data jan_feb;
merge jan (in=j rename=(weight=wt1))   ❶
      feb (in=f rename=(weight=wt2));
by ssn;
if j and f; ❷
diff = wt2 - wt1;
run;
```

In addition to the RENAME data set option, an IN data set option is use ❶. The IN data set option creates a new variable that exists only for the duration of the data step. The name assigned to the variable (after IN=) must conform to the rules for naming variables in SAS. The variable takes on either of two values, 0 or 1. If the data set contributed any information to the observation currently being created by the data step, the value of the variable is 1, otherwise it is 0. When performing a match-merge, the observation being created on each pass through the data step in example 5 will have information contributed by both data set JAN and FEB if there is a match based on the values of the by-variable SSN. This condition is checked in the subsetting IF statement ❷. The statement...

```
if j and f;
```

is equivalent to the statement...

```
if j eq 1 and f eq 1;
```

Either statement is true only when both data set JAN and data set FEB have contributed information to the an observation and only matched observations are added to the data set JAN_FEB.

```
Obs        ssn        wt1     zip     wt2    diff
 1     001-00-1234    180    12203    160     -20
 2     123-45-6789    150    13502    145      -5
 3     987-65-4321    120    12345    125       5
```

The data step can be modified to create three data set: one with matched observations; one with unmatched observations from data set JAN; one with unmatched observations from data set FEB. Once again, this is accomplishes using the IN data set option.

```
* EXAMPLE 6;
data jan_feb only_jan only_feb; ❶
merge jan (in=j rename=(weight=wt1))
      feb (in=f rename=(weight=wt2));
by ssn;
diff = wt2 - wt1;
if j and f then output jan_feb; ❷
else
if j then output only_jan;
else      output only_feb;
run;
```

Three data sets are to be created with the data step ❶. An IF-THEN-ELSE statement checks the values of the IN data set option variables and directs observations to the appropriate data set. Notice that the position of the statement that computes the difference in weights is in a different position in example 6 than it was in example 5. Do you know why it was moved?

Problem #2...you have two SAS data sets. One has demographic data on a group of individuals. The other has medical data. You want to create a single data set containing the demographic and medical data for each a single observation. The following data step creates the data sets use in the examples.

```
* EXAMPLE 7;
data demographic;
input name : $5. age zip : $5.; ❶
datalines;
ADAMS  20 12203
BROWN  21 10001
SMITH  50 12005
SMITH  33 12012
;
run;

data medical;
input name : $5. age hr chol ; ❷
label
hr   = 'heart rate'
chol = 'cholesterol'
;
datalines;
ADAMS  20 89 200
BROWN  21 60 140
SMITH  34 71 150
;
run;
```

A variable that is common to both data set is NAME ❶ ❷. Notice that there are repeated values of the variable NAME in the DEMOGRAPHIC data set, there are two people named SMITH. If these two data sets are combined with a match-merge using NAME as the by-variable, this is known as a ***many-to-one*** (or ***one-to-many*** depending on the data set that appears first in the merge statement) match-merge.

```
* EXAMPLE 8;
data both;
merge demographic medical;
by name;
run;
```

The many-to-one merge produces the following...

```
Obs    name     age     zip     hr    chol
 1     ADAMS     20    12203    89     200 ❶
 2     BROWN     21    10001    60     140
 3     SMITH     34    12005    71     150 ❷
 4     SMITH     33    12012    71     150
```

Both ADAMS and BROWN are one-to-one matches and each contributed one observation to data set BOTH ❶. However, each SMITH in data set DEMOGRAPHIC was merged with the one SMITH in data set MEDICAL, producing two observations in data set BOTH ❷.

Notice that since the variable AGE has the same name in both data sets, the value of age in data set BOTH for most of the observations comes from data set MEDICAL, the last data set mentioned in the MERGE statement. However, the age in the last observation comes from data set DEMOGRAPHIC. Understanding why this happens for the last observation has to do with how SAS forms observations when performing a match-merge with by-variables. After SAS merges the first SMITH in data set DEMOGRAPHIC with the SMITH in data set MEDICAL, there are no more SMITHs left in the MEDICAL data the next time SAS checks to see if there are any more observations left in the current by-group, that is by-group SMITH. There is one in data set DEMOGRAPHIC, so SAS reads that observation and the value of age from data set DEMOGRAPHIC replaces the 34 that was in place the last time SAS created a merged observation. There is no SMITH left in data set MEDICAL, so that value remains in the new observation.

Another item to remember is that the values of all variables read with a MERGE statement are automatically retained (as with a SET statement). If there is a BY statement in the data step (as there is with a match-merge), the values are only retained as long as there are observations left in a by-group. They are set to missing once a new by-group is encountered.

Since the variable AGE is present in both the DEMOGRAPHIC and MEDICAL data sets, you can add a statement to the data step that specifies to output only observations where the age in two data sets differs by one year or less.

```
* EXAMPLE 9;
data both;
merge demographic (rename=(age=age1)) medical (rename=(age=age2)); ❶
by name;
if  abs(age1 - age2) le 1; ❷
run;
```

The variable AGE is renamed in both data sets ❶. The renamed variables are used is a subsetting IF statement ❷. The difference between the two ages is calculated and the absolute value function (ABS) converts any negative value to positive. If the two ages are within one year, an observation is output.

```
Obs    NAME     AGE1     ZIP     AGE2    HR    CHOL
 1     ADAMS     20     12203     20     89     200
 2     BROWN     21     10001     21     60     140
 3     SMITH     33     12012     34     71     150 ❶
```

The subsetting IF statement eliminated one of the SMITH matches and the one kept in the data set merged two observations with ages within one year of each other ❶.

Problem #3...a problem identical to problem #2, except the data sets are larger. You have two SAS data sets. One has a name and an age. The other has a name, an age, and a heart rate. You want to create a single data set containing the name, age from both data sets, and the heart rate. The following data step creates the data sets use in the examples.

```
data dataset1;
input name $ age; ❶
datalines;
ADAMS    20
BROWN    21
BROWN    30
JONES    45
JONES    46
JONES    47
LAWRENCE 10
LAWRENCE 14
LAWRENCE 16
SMITH    50
WALTERS  29
;
run;

data dataset2;
input name $ age hr; ❷
label hr = 'heart rate';
datalines;
ADAMS     21  89
BROWN     15  60
BROWN     21  75
BROWN     40  80
JONES     48  60
KELLY     57  90
LAWRENCE  16  60
LAWRENCE  20  84
SMITH     30  71
SMITH     50  55
;
run;
```

Once again, a variable that is common to both data set is NAME ❶ ❷. Notice that there are repeated values of the variable NAME in the both data sets, not just one as in problem. If these two data sets are combined with a match-merge using NAME as the by-variable, this is known as a ***many-to-many*** match-merge. The two data sets also contain observations that will be used in ***one-to-one***, ***one-to-many***, and ***many-to-one*** match merges. There are also observations in both data sets that will be unmatched. The SAS code from example 9 is used again.

```
* EXAMPLE 11;
data both;
merge
dataset1 (rename=(age=age1))
dataset2 (rename=(age=age2))
;
by name;
if  abs(age1 - age2) le 1;
run;
```

and some of the matches that should be in the data set BOTH (for example BROWN who is 21 in DATASET1 with BROWN who is 21 in DATASET2) have not been output.

```
Obs     NAME      AGE1    AGE2    HR
 1     ADAMS       20      21     89 ❶
 2     JONES       47      48     60 ❷
 3     KELLY        .      57     90 ❸
 4     SMITH       50      50     55 ❹
 5     WALTERS     29       .      . ❸
```

ADAMS is a one-to-one match ❶. JONES is a many-to-one match ❷. KELLY and WALTERS are unmatched, but are in data set BOTH since the default result of a match-merge is to include all unmatched observations in the output data set ❸. SMITH is one-to-many match ❹. The only matches that should be in data set BOTH but are not are the many-to-many matches. Eliminating the subsetting IF statement shows why the many-to-many matches are missing.

```
Obs    NAME        AGE1    AGE2    HR
  1    ADAMS        20      21     89
  2    BROWN        21      15     60 ❶
  3    BROWN        30      21     75
  4    BROWN        30      40     80
  5    JONES        45      48     60
  6    JONES        46      48     60
  7    JONES        47      48     60
  8    KELLY         .      57     90
  9    LAWRENCE     10      16     60 ❷
 10    LAWRENCE     14      20     84
 11    LAWRENCE     16      20     84
 12    SMITH        50      30     71
 13    SMITH        50      50     55
 14    WALTERS      29       .      .
```

There is a 21 year old BROWN in both data sets that should match ❶. However, you can see in the above listing that the two observations that should match never were matched in the many-to-many match merge. The same is true for the 16 year old LAWRENCE in both data sets ❷. Whenever you perform a many-to-many match merge, you will see the following NOTE in the LOG...

```
NOTE: MERGE statement has more than one data set with repeats of BY values.
```

This is an indication that the data set produced by the match-merge may or may not be the one that should be produced based on your SAS code. In example 11, it clearly was not. PROC SQL is an alternative to a match-merge that will properly handle all matching situations.

```
* EXAMPLE 12;
proc sql; ❶
   create table both as ❷
   select dataset1.name, dataset1.age as age1, dataset2.age as age2, hr ❸
   from dataset1, dataset2 ❹
   where dataset1.name eq dataset2.name and
   abs(age1-age2) le 1; ❺
quit; ❶
```

If you are new to PROC SQL, no data sets are specified when the procedure is invoked (though there a numerous options that can be specified to control both the output and PROC SQL execution) and the procedure ends with a QUIT statement ❶. SAS data sets created by PROC SQL are referred to as tables ❷. The variables to be included in the table (data set) are specified in a SELECT clause ❸. Notice that variable names are separated by commas, something that is not normally done in SAS outside of PROC SQL. Also, in the data step match-merge, the problem of variable names common to two data sets was handled using the RENAME data set option. In PROC SQL, you specify both the data set name and variable concatenated by a period to distinguish between the variables with common names in the two data sets. A FROM clause names the data sets that are to be joined (merged) ❹. Merely separating the data set names by a comma results in what is referred to in PROC SQL as an inner-join. Finally, a WHERE clause specifies the rules for combining the data sets mentioned in the FROM clause ❺. PROC SQL code used in example 12 produces the following.

```
Obs    NAME        AGE1    AGE2    HR
 1     ADAMS        20      21     89
 2     BROWN        21      21     75
 3     JONES        47      48     60
 4     LAWRENCE     16      16     60
 5     SMITH        50      50     55
```

Notice that all the matches are in the data set (one-to-one, one-to-many, many-to-one, many-to-many). The many-to-many matches appear since PROC SQL merges observations differently from a match-merge. In theory, each observation in DAASET1 'sees' each observation in DATASET2 during the joining (merging) process. Once the observations are combined, the conditions in the WHERE clause are checked to see if the observation should be kept. This process results in each BROWN in DATASET1 'seeing' each BROWN in DATASET2 (same for LAWRENCE). This results in the 21 year old BROWN (and 16 year old LAWRENCE) being output to data set BOTH. No unmatched observations are in the data set. When an inner-join is performed in PROC SQL, only matched observations are output.

The type of join used in PROC SQL allows you to control the type of observations that are placed into the combined data set, performing the same function as the IN= variables in a data step merge. If two data sets are being combined with PROC SQL, different joins result in the following combined data sets.

TYPE OF JOIN	OBSERVATIONS IN COMBINED DATA SET
inner	only matched
full	all matched plus all unmatched from both data sets
left	all matched plus all unmatched from data set mentioned first
right	all matched plus all unmatched from data set mentioned last

The FULL, LEFT, and RIGHT joins are known as OUTER joins and can only be used when combining two data sets. The type of expression needed to create multiple data sets when using PROC SQL to combine data sets is not as straight forward as the process of using IN= variables with a data set merge. If you want to put all the matched observations into one data set and unmatched observations into another, you will have to learn a little more about PROC SQL.

REFERENCES (ALL HYPERLINKED)

The following papers published in the proceedings of various user group conferences contain more information on the topics covered in this paper. They are by no means exhaustive of all the papers that are available on these topics, but the listed papers will give you some more insight on both reshaping and combining data using SAS.

RESHAPING

Stuelpner, Janet. 2006. "The TRANPOSE Procedure or How to Turn It Around." *Proceedings of the Thirty-First Annual SAS Users Group International Conference*. San Francisco, CA.

Lavery, Russell. 2005. "An Animated Guide: Proc Transpose." *Proceedings of the Eighteenth Annual Northeast SAS Users Group*. Portland, ME.

Whitlock, Ian. 2000. "A Double Transpose." *Proceedings of the Thirteenth Annual Northeast SAS Users Group*. Philadelphia, PA.

Virgile, Bob. 1999. "Changing the Shape of Your Data: PROC TRANSPOSE vs. Arrays." *Proceedings of the Twenty-Fourth Annual SAS Users Group International Conference*. Miami Beach, FL.

COMBINING
Defoor, Jimmy. 2006. "Proc SQL – A Primer for SAS® Programmers." *Proceedings of the Thirty-First Annual SAS Users Group International Conference*. San Francisco, CA.

(note...the above paper includes examples that compare data step merges with merging of data sets using PROC SQL)

Loren, Judy, and Gaudana, Sandeepl. 2005. "Join, Merge, or Lookup? Expanding Your Toolkit." *Proceedings of the Eighteenth Annual Northeast SAS Users Group*. Portland, ME.

Schreier, Howard. 2005. "Let Your Data Power Your DATA Step: Making Effective Use of the SET, MERGE, UPDATE, and MODIFY Statements." *Proceedings of the Thirtieth Annual SAS Users Group International Conference*. Philadelphia, PA.

Foley, Malachey. 2005. "MERGING vs. JOINING: Comparing the DATA Step with SQL." *Proceedings of the Thirtieth Annual SAS Users Group International Conference*. Philadelphia, PA.

Virgile, Bob. 2003. "Danger: MERGE Ahead! Warning: BY Variable with Multiple Lengths!" *Proceedings of the Twenty-Eight Annual SAS Users Group International Conference*. Seattle, WA.

Foley, Malachey. 1997. "Advanced MATCH-MERGING: Techniques, Tricks, and Traps." *Proceedings of the Twenty-Second SAS Users Group International Conference*. San Diego, CA.

ACKNOWLEDGMENTS

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are registered trademarks or trademarks of their respective companies.

CONTACT INFORMATION

The author can be contacted using e-mail... msz03@albany.edu