Paper 109-25

Merges and Joins

Timothy J Harrington, Trilogy Consulting Corporation

Abstract

This paper discusses methods of joining SAS® data sets. The different methods and the reasons for choosing a particular method of joining are contrasted and compared. Potential problems and limitations when joining data sets are also discussed.

The need to combine data sets constantly arises during software development, as does the need to validate and test new code. There are two basic types of join, *vertical*, and *horizontal.* Vertical joining is appending one data set to another, whereas horizontal joining is using one or more *key variables* to combine different observations.

Vertical Joining

A good example of vertical joining is adding to a data set in time sequence, for example, adding February's sales data to January's sales data to give a year-to-date data set. Providing both data sets have the same variables and all the variables have the same attributes such as data type, length, and label, there is no problem. However, once the data sets are combined at least one of the variables should, in practice, be able to identify which of the source data sets any given observation originated from. In this sales data example a date or month name should be present to indicate whether a given observation came from January's data or February's data. Another issue may be the sort order. In this example there is no need to sort the resulting data set if the source data sets are in date order, but if, say, the data sets were sorted by product code, or sales representative the resulting data set would need to be resorted by date. Most importantly, when vertically joining data sets, is the issue vertical compatibility. This is whether the corresponding variables in each data set have the same attributes, and if there are any variables which are present in one data set but not in the other.

Using PROC DATASETS and APPEND

One method of vertical joining is to use the utility procedure PROC DATASETS with the APPEND statement. More than two data sets many be joined in this way, but all of the data sets should be vertically compatible. However, vertical incompatibility may be overridden by using the FORCE option. When this option is used, variables which are absent in one data set are created with the same attributes in the resulting data set, but the values are missing in each observation which originated from the data set without those variables. Where variable lengths are different the shorter length values are right padded with spaces to equal the longer length. Where data types are different the numeric type is made character. If labels are different the label from the latest data set is used. If the FORCE option is not specified and any of the data sets are not completely vertically compatible applicable NOTES and WARNINGS are written to the log file. If a variable is present in the DATA data set but is absent in the BASE data set the appending is not done. The example below appends two data sets DATA01 and DATA02 to the data set DATA99. DATA99 is the 'Base' data set, which, if it does not exist is created and becomes the compound of DATA01 and DATA02 (A NOTE of this is written to the Log file). The NOLIST option in PROC DATASETS prevents it from running interactively.

```
PROC DATASETS NOLIST;
  APPEND BASE= DATA99 DATA= DATA01
  APPEND BASE= DATA99 DATA= DATA02;
RUN;
```

If observation order is important after appending, a PROC SORT should be performed on the compound data set (DATA99 in this example) by the appropriate BY variables.

Vertical Joining with UNION Corresponding

In PROC SQL two or more data sets may be vertically joined used UNION CORRESPONDING ALL. (If the 'ALL' is omitted only one of any duplicate observations are kept). This is analogous to APPEND in PROC DATASETS but if the data sets to be joined are not vertically compatible only variables common to both data sets are placed in the resulting table. This is the same example as above, but using PROC SQL with UNION CORRESPONDING ALL.

```
PROC SQL;
  CREATE TABLE DATA99 AS
    SELECT *
      FROM DATA01
  UNION CORRESPONDING ALL
    SELECT *
      FROM DATA02;
QUIT;
```

This PROC SQL works if DATA99 is being created as new, but if DATA99 already exists and the intention is append DATA01 and DATA02 to this data set the code must be written as

```
PROC SQL;
  CREATE TABLE DATA99 AS
    SELECT *
      FROM DATA99
  UNION CORRESPONDING ALL
    SELECT *
      FROM DATA01;
  UNION CORRESPONDING ALL
    SELECT *
      FROM DATA02;
QUIT;
```

Generally this method is less efficient than using PROC DATASETS with APPEND.

Horizontal Joining

There are four basic types of horizontal join, the *inner join*, *left join*, *right join*, and *full join*. All such joins are *Cartesian products* made on specified key variables. If there are duplicate matches in either or both tables *all* of the matching observations are selected, for example if there are two equal key values in each input data set there will be four output observations created.

The following example data sets are being used to demonstrate horizontal joins. These data sets called DOSING and EFFICACY are hypothetical clinical trials data sets. In the DOSING data set PATIENT is the patient id number, MEDCODE is the test medication (A or B), DOSE_ID is an observation id number, DOSEAMT is the amount of dose in mg, and DOSEFRQ is the dose frequency in doses per day. The EFFICACY data set contains an observation id number, EFFIC_ID, a VISIT number, and an efficacy SCORE (1 to 5). The variables DOSE_ID and EFFIC_ID in this example are for easy identification of the data set and input observation which contributed to the resulting output observation.

```
The DOSING data set

             MED          DOSE  DOSE
OBS  PATIENT CODE DOSE_ID AMT   FRQ

1      1001   A      1     2     2
2      1003   A      2     1     2
3      1004   A      3     1     2
4      1004   B      4     4     2
5      1006   B      5     2     2
6      1007   A      6     2     1
7      1008   A      7     1     2
8      1009   A      8     2     2
```

```
The EFFICACY data set

OBS  PATIENT EFFIC_ID VISIT SCORE

 1     1001     1        1     4
 2     1002     2        1     5
 3     1004     3        1     2
 4     1004     4        2     1
 5     1005     5        1     2
 6     1009     6        1     5
```

The Inner Join

The inner join creates observations from data items selected from either input data set where the key values match in *both* tables. If the key values match in only one table an output observation is *not* created. An 'inner' join is a logical AND of the two tables and is therefore commutative, that is the tables can be joined in either order. The following PROC SQL segment creates a table named INNER1 as the inner join between DOSING and EFFICACY on PATIENT. A point to note is that where there are duplicate key values a complete Cartesian product is produced, in this example this happens with Patient 1004. The 'A' and 'B' characters preceding the variable names are aliases for each of the data set names and the ORDER BY clause sorts the resulting data set in ascending order of PATIENT and MEDCODE.

```
Table INNER1: An INNER JOIN on PATIENT
between DOSING and EFFICACY.

PROC SQL;
  CREATE TABLE INNER1 AS
    SELECT A.*, B.EFFIC_ID, B.VISIT,
       B.SCORE
       FROM DOSING A, EFFICACY B
          WHERE A.PATIENT=B.PATIENT
            ORDER BY PATIENT;
QUIT;
```

```
 PAT- MED  DOSE DOSE          DOSE EFFIC
 IENT CODE AMT  FRQ VISIT SCORE ID   ID

1 1001  A    2    2   1     4    1    1
2 1004  A    1    2   1     2    3    3
3 1004  B    4    2   1     2    4    3
4 1004  B    4    2   2     1    4    4
5 1004  A    1    2   2     1    3    4
6 1009  A    2    2   1     5    8    6
```

This resulting table, INNER1, contains only observations with Patient Numbers common to *both* data sets. There are four observations for Patient 1004 because of the Cartesian product of two observations with this Patient Number in each data set. If the WHERE clause were omitted the complete Cartesian product of every observation would be selected, producing 48 (6*8) observations, hence at least one key variable must be specified when performing any type of horizontal join on data sets of more than a few observations. Another point to note is that instead of using the WHERE clause, the FROM statement could be rewritten as FROM DOSING A INNER JOIN EFFICACY B.

The Left Join

A 'left' join selects items from all the observations in the first (left) data set regardless of their key values but only observations with matching key values from the second (right) data set. Variables from the second data set where the observation key values do not match the join criteria are assigned missing values in the output data set. In this example the table LEFT1 is created by performing a left join on DOSING and EFFICACY. As with the inner join, where there are multiple key values a complete Cartesian product is created. Points to note are that non-key items from the second data set (EFFICACY) are missing where there is no key match, as in observations 2, 7, 8, and 9. Also, a left join is not commutative, reversing the order of the source data sets would produce a totally different result.

Table LEFT: A LEFT JOIN on PATIENT between DOSING ('left' data set) and EFFICACY ('right' data set).

```
PROC SQL;
  CREATE TABLE LEFT1 AS
    SELECT A.*, B.EFFIC_ID, B.VISIT,
      B.SCORE
      FROM DOSING A LEFT JOIN EFFICACY B
        ON A.PATIENT = B.PATIENT
          ORDER BY PATIENT;
QUIT;
```

```
  PAT- MED  DOSE DOSE         DOSE EFFIC
  IENT CODE AMT  FRQ VISIT SCORE ID   ID

 1 1001   A    2    2   1     4    1    1
 2 1003   A    1    2   .     .    2    .
 3 1004   A    1    2   1     2    3    3
 4 1004   B    4    2   1     2    4    3
 5 1004   A    1    2   2     1    3    4
 6 1004   B    4    2   2     1    4    4
 7 1006   B    2    2   .     .    5    .
 8 1007   A    2    1   .     .    6    .
 9 1008   A    1    2   .     .    7    .
10 1009   A    2    2   1     5    8    6
```

The Right Join

A 'right' join is where all the observations are selected from the second data set and where observations do not match in the first data set the key values themselves are assigned missing values. A right join, like a left join, is not commutative neither is a right join the same as reversing the order of the two data sets in a left join. In this example observations 1 and 2 have missing values for PATIENT and the non-key items from the DOSING data set because two observations in EFFICACY, with PATIENT numbers 1002 and 1005, do not have the same patient numbers as any of the observations in DOSING. (The ORDER BY PATIENT clause causes the missing values to float to the topmost observations.).

Table RIGHT1: A RIGHT JOIN on PATIENT between DOSING('left' data set) and EFFICACY('right' data set).

```
PROC SQL;
  CREATE TABLE RIGHT1 AS
    SELECT A.*, B.EFFIC_ID, B.VISIT,
      B.SCORE
      FROM DOSING A RIGHT JOIN
        EFFICACY B
        ON A.PATIENT = B.PATIENT
          ORDER BY PATIENT;
QUIT;
```

```
 PAT- MED  DOSE DOSE         DOSE EFFIC
 IENT CODE AMT  FRQ VISIT SCORE ID   ID

1    .       .    .   1     5   .    2
2    .       .    .   1     2   .    5
3 1001  A    2    2   1     4   1    1
4 1004  B    4    2   2     1   4    4
5 1004  B    4    2   1     2   4    3
6 1004  A    1    2   2     1   3    4
7 1004  A    1    2   1     2   3    3
8 1009  A    2    2   1     5   8    6
```

The Full Join

The 'full' join selects all the observations from both data sets but there are missing values where the key value in each observation is found in one table only. A 'full' join is the logical OR of the two tables, but is not commutative because missing key values are assigned to those non-matching observations in the second data set. For example, if the order of the data sets was reversed the missing values of PATIENT would be due to PATIENTs 1003, 1006, 1007, and 1009 being in DOSING and not in EFFICACY, instead of being due to PATIENTs 1002 and 1005 being in EFFICACY and not in DOSING.

Table FULL1: A FULL JOIN on PATIENT between DOSING('left' data set) and EFFICACY('right' data set).

```
PROC SQL;
  CREATE TABLE FULL1 AS
    SELECT A.*, B.EFFIC_ID, B.VISIT,
      B.SCORE
      FROM DOSING A FULL JOIN
        EFFICACY B
        ON A.PATIENT = B.PATIENT
          ORDER BY PATIENT;
QUIT;
```

```
 PAT- MED  DOSE DOSE         DOSE EFFIC
 IENT CODE AMT  FRQ VISIT SCORE ID   ID

 1    .      .    .   1     5   .    2
 2    .      .    .   1     2   .    5
 3 1001 A    2    2   1     4   1    1
 4 1003 A    1    2   .     .   2    .
 5 1004 B    4    2   1     2   4    3
 6 1004 B    4    2   2     1   4    4
 7 1004 A    1    2   1     2   3    3
 8 1004 A    1    2   2     1   3    4
 9 1006 B    2    2   .     .   5    .
10 1007 A    2    1   .     .   6    .
11 1008 A    1    2   .     .   7    .
12 1009 A    2    2   1     5   8    6
```

Using the COALESCE function and determining the source data set

When performing left, right, or full joins where observations do not have a key variable match, non-key values are assigned missing values. Sometimes there is a need to substitute missing values with other data, either hard coded values or different items from either data set. One way to do this is with a CASE construct, but the COALESCE function is provided specifically for this purpose. In this example the variable ADJSCORE (Adjusted Score) contains the value of SCORE in the observations that match, but where there is no match and SCORE is missing AJDSCORE is assigned the value zero. If a value of SCORE was missing in a matching observation the value of ADJSCORE would also be set to zero. COALESCE may be used with either a character or numeric data type, but the second argument must be of that same data type. The CASE statement in the example assigns the values ‘Match’ or ‘Miss’ to the character variable INVAR depending on whether the values of PATIENT match or not.

COALESCE function example

```
PROC SQL;
  CREATE TABLE LEFT1A(DROP=DOSE_ID) AS
    SELECT A.*, B.VISIT, B.SCORE,
      COALESCE(B.SCORE,0) AS ADJSCORE,
      CASE (A.PATIENT=B.PATIENT)
        WHEN 1 THEN ‘Match’
        WHEN 0 THEN ‘Miss’ ELSE ‘ ‘
      END AS INVAR LENGTH=5
      FROM DOSING A LEFT JOIN
        EFFICACY B
        ON A.PATIENT = B.PATIENT
          ORDER BY PATIENT, MEDCODE;
QUIT;
```

```
 PAT- MED DOSE DOSE            ADJ   IN-
 IENT CODE AMT FRQ VISIT SCORE SCORE VAR

 1 1001 A   2    2   1     4     4 Match
 2 1003 A   1    2   .     .     0 Miss
 3 1004 A   1    2   1     2     2 Match
 4 1004 B   4    2   1     2     2 Match
 5 1004 A   1    2   2     1     1 Match
 6 1004 B   4    2   2     1     1 Match
 7 1006 B   2    2   .     .     0 Miss
 8 1007 A   2    1   .     .     0 Miss
 9 1008 A   1    2   .     .     0 Miss
10 1009 A   2    2   1     5     5 Match
```

The Data Step Merge

A DATA step MERGE joins two data sets inside a DATA step to produce a resulting data set. A DATA step MERGE differs from a PROC SQL in two important ways. (1) The sort order of the key variables is important because matching is performed sequentially on an observation-by-observation basis. (2) Cartesian products are not evaluated, the merge is performed sequentially by input observation and then the resulting observation is placed in the Program Data Vector (PDV). When there are more key matching observations in one data set than the other, the non-key data from the last of the fewer matching observations is retained in the PDV for each remaining match of the more numerous observations. This is called the *implied retain.*

Sort Ordering and BY Variables

BY variables are the ‘key’ variables on which the merge matching is to be performed. They must have been sorted in either ascending (default) order or descending order using PROC SORT or an ORDER BY statement in a prior PROC SQL. If the BY variables are not sorted, or are sorted in an inappropriate order, in either of the input data sets this error results:

```
ERROR: BY variables are not properly
sorted on <Data Set name>.
```

The DATA step performing the merge must contain an applicable BY statement, matching the BY statement in the preceding PROC SORT or PROC SQL ORDER BY statement of each corresponding set. If a key variable has been sorted in descending order that variable must be specified as DESCENDING in the BY statement of the merge.

If the BY values are unique in both data sets the merge is a ‘one to one’ merge. If there are observations with duplicate BY values in one data set and only one matching observation in the other data set that single observation is joined with all of the matching BY variables in the first data set because of the implied retain. This is a ‘one to many’ merge. This aspect of merging is commutative, in that performing a ‘many to one’ merge with the order of the data sets reversed produces the same result. A ‘Many to many’ merge is where there are corresponding duplicate BY variables in both data sets. Such a merge does not result in a Cartesian product because the observations are joined in sequence where they match. (See what happens to Patient 1004 in the examples listed below). When ‘many to many’ situations are encountered the following message is written to the log file:

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

Merges and IN Variables

An IN variable is a Boolean flag, which applies to an input data set. The IN variable is set to ‘true’ (1) or ‘false’ (0) depending on whether that data set contributed data to the current PDV contents or not. When two data sets are being merged at least one of the IN variables must be ‘true’, both IN variables are ‘true’ if the BY variables match. IN variables are most useful for testing for such matches, as shown in the examples listed below.

Comparing a DATA Step Merge with a PROC SQL Join.

The following examples use a DATA step MERGE instead of a PROC SQL join to perform corresponding joins on the same key variables as shown above using the DOSING and EFFICACY data sets.

This first example performs a merge using the key variable PATIENT and outputs an observation when the patient numbers are equal, and hence the IN variables are both true. This merge corresponds to the PROC SQL inner join, but with one important difference, no Cartesian products are generated because the merging process is sequential by observation. Hence, the data set INNER2 has only two observations for Patient 1004 instead of the four in INNER1. (For this reason ‘many to many’ merges should be avoided in practice). Another point to note is that when testing IN variables an IF statement must be used, a WHERE clause will not work because the IN variables are calculated within the DATA Step and are not from the source data.

Table INNER2: A MERGE between DOSING and EFFICACY where equal values of PATIENT occur in both input data sets.

```
DATA INNER2;
  MERGE DOSING(IN=A) EFFICACY(IN=B);
  BY PATIENT;
  IF (A=B);
RUN;
```

```
 PAT- MED  DOSE DOSE           DOSE EFFIC
 IENT CODE AMT  FRQ VISIT SCORE ID   ID

1 1001  A    2    2   1     4   1    1
2 1004  A    1    2   1     2   3    3
3 1004  B    4    2   2     1   4    4
4 1009  A    2    2   1     5   8    6
```

This second example performs the same merge as the first example but only outputs an observation whenever the DOSING data set contributes a value to the PDV. Observations with values of PATIENT which are present only in EFFICACY and not in DOSING are not output and missing values are substituted in the non-key variables not from DOSING. This merge corresponds to the PROC SQL left join, but without the Cartesian product of duplicate key values (Patient 1004).

Table LEFT2: All values of PATIENT are taken from DOSING and only matching values from EFFICACY.

```
DATA LEFT2;
  MERGE DOSING(IN=A) EFFICACY;
  BY PATIENT;
  IF A;
RUN;
```

```
 PAT- MED  DOSE DOSE             DOSE EFIC
 IENT CODE AMT  FRQ VISIT SCORE ID   ID

1 1001  A    2    2   1     4    1    1
2 1003  A    1    2   .     .    2    .
3 1004  A    1    2   1     2    3    3
4 1004  B    4    2   2     1    4    4
5 1006  B    2    2   .     .    5    .
6 1007  A    2    1   .     .    6    .
7 1008  A    1    2   .     .    7    .
8 1009  A    2    2   1     5    8    6
```

In this third example only observations in the PDV from the EFFICACY data set are output. Values in DOSING which do not match, including the key value (Patient) are output as missing values. This corresponds to a right join in PROC SQL. Note that both the order in which the data sets are specified and which IN variable is tested are important. In this example ‘merge dosing efficacy(in=b)’ is not the same as ‘merge efficacy(in=b) dosing’ or ‘merge dosing(in=b) efficacy’.

Table RIGHT2: A MERGE between DOSING and EFFICACY where all values of PATIENT are taken from EFFICACY and only matching values from DOSING.

```
DATA RIGHT2;
  MERGE DOSING EFFICACY(IN=B);
  BY PATIENT;
  IF B;
RUN;
```

```
 PAT- MED  DOSE DOSE           DOSE EFFIC
 IENT CODE AMT  FRQ VISIT SCORE ID   ID

1 1001 A    2    2    1    4    1    1
2 1002      .    .    1    5    .    2
3 1004 A    1    2    1    2    3    3
4 1004 B    4    2    2    1    4    4
5 1005      .    .    1    2    .    5
6 1009 A    2    2    1    5    8    6
```

In this next and final example all observations in the PDV are output regardless of a match or not, hence IN variables are not needed. This corresponds to the PROC SQL full join.

Table FULL2: A MERGE between DOSING and EFFICACY where all values of PATIENT are taken from both data sets regardless of whether they match or not.

```
DATA FULL2;
  MERGE DOSING EFFICACY;
  BY PATIENT;
RUN;
```

```
 PAT- MED  DOSE DOSE           DOSE EFFIC
 IENT CODE AMT  FRQ VISIT SCORE ID   ID

 1 1001 A    2    2   1     4   1    1
 2 1002      .    .   1     5   .    2
 3 1003 A    1    2   .     .   2    .
 4 1004 A    1    2   1     2   3    3
 5 1004 B    4    2   2     1   4    4
 6 1005      .    .   1     2   .    5
 7 1006 B    2    2   .     .   5    .
 8 1007 A    2    1   .     .   6    .
 9 1008 A    1    2   .     .   7    .
10 1009 A    2    2   1     5   8    6
```

PROC SQL Joins vs. DATA Step MERGEs

There are several important differences between PROC SQL joins and DATA Step MERGES. A SQL join creates a Cartesian product out of multiple occurrences of key values. When there are more matching key observations in one data set than in the other ('one to many' or 'many to one' merges) the contents of the last matching observation from the data set with fewer matches are retained in the PDV. This is the implied retain. The result is the remaining non-key values from the data set with fewer matches appear in the corresponding excess output observations. The implied retain does not occur in the above examples with Patient 1004 because there are the same number of observations for this patient in both data sets. If one of the Patient 1004 observations is deleted from either DOSING or EFFICACY an implied retain would then take place with the second observation. A 'many-to-many' MERGE does not produce a complete Cartesian product with duplicate key values in both data sets ('many to many'). A note indicating repeating BY values is written to the log file. 'Many-to-many' merges are also expensive in terms of processing time and resources. Hence 'many-to-many' merges should be avoided.

When using a SQL join the observations in either data set do not have to be sorted, a DATA step MERGE requires the key variables (BY variables) to have been sorted in a corresponding order, either using PROC SORT, or an ORDER BY in a prior PROC SQL.

A PROC SQL join can use aliases to identify the data set, which is contributing a particular variable. A DATA Step MERGE can be used with a logical IN variable to identify which data set contributed the key values to the current PDV contents. Specific non-key variable names must be unique to each data set since their source data set cannot be identified with an alias. Data set options such as KEEP, DROP, and RENAME may be used in both PROC SQL or in a DATA step. To subset data from either input data set a WHERE statement may be used in parenthesis after the data set name (This is more efficient than using the WHERE statement in a CREATE TABLE block or a DATA step.)

Runtime benchmark tests show that a PROC SQL join is faster than a DATA Step MERGE. Using indexes improves performance still further, but PROC SQL indexes and DATA Step indexes are implemented differently internally by the SAS system and may conflict with each other curtailing performance.

More than Two Data Sets at a Time

In any of the examples shown above more than two data sets may be joined. However, due to the increased complexity, such as possible large Cartesian products when performing PROC SQL joins, joining more than two data sets at a time should be avoided. When the requirement is to join many data sets a 'result' data set should be created from a join of the first two and then this 'result' data set be joined with each other data set one at a time.

References

Timothy J. Harrington, Trilogy Consulting
Corporation, 5148 Lovers Lane, Kalamazoo, MI
49002. (616) 344 1996.
TJHARRIN@TRILOGYUSA.COM