Paper 87-27

Merging Flat Files Directly

Robert Patten, Lands' End, Dodgeville, WI

Abstract

Typically, pulling data together from various flat file sources requires the following steps: 1. read each source into a separate SAS® dataset; 2. sort, if necessary; 3. merge these datasets into a single dataset. Eliminating the need to create intermediate SAS datasets, especially when dealing with large multiple flat files, can reduce run times substantially. As an example, I have applied this technique in a job that runs in a production environment on a weekly basis. The job, which previously ran in eighteen hours now runs in under six hours! Merging flat files requires that these files be sorted into some common key value. This is not as tight a restriction as it first appears. External data often is stored in some key value order. Many times we have some control over this order. Also, you could sort the files prior to processing.

Introduction

When you are in a situation that requires you to merge data from multiple external sources (more specifically flat files or text files), what do you do? You might immediately think of the following steps:

1. Read each external file into an intermediate SAS dataset
2. Sort these intermediate datasets into key order, if needed
3. Merge these intermediate datasets by key order into the final dataset needed

Depending on the size of the external data files, these steps can be extremely time and resource hungry. Wouldn't it be nice if you could skip the first two steps? You can bypass them if the external files were already sorted by the key values. This is not as unusual as you might expect. External data often is stored in some key value order. Many times you have control over this order. For example, with vendor-supplied external data you could request that the data be provided already sorted in some key order. The trick is to ensure that the files you wish to merge are sorted in the same order, then make use of this fact directly in the merging process.

The following chart illustrates the potential time savings from merging flat files directly verses pulling each flat file into an intermediate SAS dataset then merging. This information assumes that all files are pre-sorted on some common key value. Also, duplicate key values are not present.

Chart 1 : Creating and merging intermediate SAS files
vs. Merging flat files directly
Real time comparison (seconds)

	Merging Intermediate Files	Merging Directly
Reading in file1	2.86	----
Reading in file2	3.25	----
Reading in file3	2.99	----
Merging	4.18	6.16
Total Seconds:	**13.28**	**6.16**

Note: timings were taken from SAS version 8 running on a PC.

Merging flat files directly can cut time substantially! This savings result is based on using only three flat files of moderate size each containing only two variables. Actual savings are dependent on many factors: number of files, numbers of records, number of variables, platform, and data profile. I have applied this technique in a job that runs in a production environment on a weekly basis. Although it is hard to quantify the impact of merging directly because of other improvements that were made at the same time, the job, which previously ran in eighteen hours wall time, now runs in under six hours!

The following example demonstrates how you can accomplish this…

Example : Merging three flat files

In this example, you want to merge three flat files. One file is designated as the driver file while the other two are support files – you will see why this is important later on. All three files are sorted by a common key field – customer number. One of the support files contains this key and household income – the income file. The other contains the key and sales – the sales file. The driver file contains the key and address. The layout of each file is:

File	Name	Type	Position	Length
Driver	CUSTNUM	Numeric	1	1
	ADDRESS	Char	3	6
Support1	KEY1	Numeric	1	1
	INCOME	Numeric	3	3
Support2	KEY2	Numeric	1	1
	SALES	Numeric	3	3

The actual data for this example is:

Obs#	Driver (CUSTNUM,ADDRESS)	Support1 (KEY1,INCOME)	Support2 (KEY2,SALES)
1	1,MAIN	2,100	1,20
2	2,FIRST	3,60	3,40
3	5,SECOND	4,140	4,100
4	---	5,240	---

Example Code

The following is the actual code that is needed to merge these files.

```
**********************************************************
** Merging flat files: SUGI presentation
;
 1  data final(keep=CUSTNUM ADDRESS INCOME SALES);
 2  retain DONE1 DONE2 0 KEY1 KEY2 ADVANCE;

    /* input driver file information */
 3   infile 'driver.txt';
 4   input @1  CUSTNUM   1.
 5         @3  ADDRESS $6.;

    /* input each supporting file information */
    /* income file first                      */
 6   if not DONE1 then do;
 7     ADVANCE=0;
 8     infile 'income.txt' end=LAST1;
 9     do until (KEY1>=CUSTNUM or LAST1);
10       if ADVANCE then input;
11       input @1 KEY1 1. @@;
12       ADVANCE=1;
13     end;
14     if KEY1=CUSTNUM then
15       input @3 INCOME  3.;
16     if LAST1 then if KEY1<=CUSTNUM then DONE1=1;
17   end;

    /* sales file next                        */
18   if not DONE2 then do;
```

```
19    ADVANCE=0;
20    infile 'sales.txt' end=LAST2;
21    do until (KEY2>=CUSTNUM or LAST2);
22      if ADVANCE then input;
23      input @1 KEY2 1. @@;
24      ADVANCE=1;
25    end;
26    if KEY2=CUSTNUM then
27      input @3 SALES 3.;
28    if LAST2 then if KEY2<=CUSTNUM then DONE2=1;
29  end;
30  run;
```

Note: the **[Line x]** notation in the remaining discussion refers directly to the line numbers in the code above.

The heart of the code is found starting in **[Line 9]** with the DO loop. The loop controls support file processing by comparing the support file key with the driver file CUSTNUM. It provides a mechanism for stepping through the support files one record at a time, checking for a match with the driver file. A key component within the loop is the use of the double trailing @@ **[Line 11]**. The double trailing @@ holds an input line not only for the current iteration of the data step but also for subsequent iterations. Lines can be held using either the single trailing @ or double trailing @@. When a line is held with a single trailing @, these lines are released at the end of the current iteration of the data step. When a line is released, the file pointer for that file is positioned at the beginning of the next input record. A non-held INPUT statement also moves the file pointer to the next record **[Line 10,15]**. The double trailing @@ keeps the file pointer on the same record even when the current iteration of the data step is done. This allows you to remain on the same record and continue to read from that record on subsequent data step iterations.

There are five possible conditions where the DO loop will be terminated:

1. LAST1=true (the file pointer is on the last record of the file)
2. LAST1=true AND KEY1=custnum
3. LAST1=true AND KEY1>custnum
4. KEY1=custnum
5. KEY1>custnum

1. If the last record of the support file has been read, you do not want to attempt to read another record. If this were to happen then the current data step would terminate because you would be attempting to read past the end of file - SAS would assume that the data step is complete and no additional records would be added to the final merged file. You want the data step to terminate only when the **driver** file has run out of records. Thankfully, when SAS reads a record, it looks ahead to the next record to determine if the record it just read is the last record in the file. If it is the last record, then SAS sets the LAST1 variable to true. DONE1 will be set depending on the KEY1 relationship to CUSTNUM **[Line 16]**. Conditions 2 and 3 below discuss when KEY1 >= CUSTNUM. This leaves KEY1 < CUSTNUM. In this case, DONE1 is set to true. This in effect turns off any further processing for this support file because **[Line 6]** condition "not DONE1" will be false. You will not perform another INPUT statement on this file therefore you cannot read past end of file.

2. What happens if you just read in the last record of the support file (LAST1=true) and KEY1 is equal to CUSTNUM? Because you read KEY1 using the double trailing @@ to hold the line, the next INPUT **[Line 15]** associated with KEY1=CUSTNUM does not try to read beyond the end of the file - it is still reading from the same last record. Since this INPUT statement does release the held line, any subsequent INPUT statements would attempt to read past the end of file. Since LAST1 is true and KEY1 = CUSTNUM, DONE1 is set to true **[Line 16]**. No further processing would take place on this support file.

3. What happens when LAST1 is set to true and KEY1 is greater than CUSTNUM? This time DONE1 is not set to true **[Line 16]**. Although you cannot read anymore records (this would be an attempt to read past the end of the support file and the data step would be terminated) you still want to continue checking for a match between KEY1 and CUSTNUM because CUSTNUM could "catch up" with KEY1 on subsequent data step iterations. The DO loop **[Line 9]** is never entered because LAST1 is set to true. KEY1 is retained **[Line 2]** so you continue to check its last value against newly input driver file CUSTNUM values **[Line 14]**. Because the last record for this support file was read in using the double trailing @@, the line will continue to be held through subsequent data step iterations until either KEY1 equals CUSTNUM or the data step processing is complete.

4. When the KEY1 value just read in is equal to CUSTNUM, the remaining information for that record is read into the program data vector **[Line 15]**. At the end of the current iteration of the data step, this information is written to the dataset being built. The input line is released and the next KEY1 value will be read during the next iteration of the data step.

5. The importance of using the double trailing @@ becomes apparent when the KEY1 value is greater than CUSTNUM. In this case, there is no subsequent INPUT statement to release the line. The line is held for the next data step iteration. Why is this important? Because when KEY1 is greater than CUSTNUM, there still remains the possibility of a match between the two. Subsequent reads of CUSTNUM could "catch up" to the current value of KEY1. If at the end of the iteration, the current held record containing the value of KEY1 were released, you would lose that potential match because on the next data step iteration a new KEY1 value would be read in.

Step by Step Analysis

Iteration 1:
Initial file pointer positions:

```
   Driver        Income       Sales
-> 1 Main     -> 2 100      -> 1 20
   2 First       3 60          3 40
   5 Second      4 140         4 100
                 5 240
```

On each iteration of the data step, the driver file information is read in first (CUSTNUM=1, ADDRESS=Main) **[Lines 3-5]**. The driver file information will always be present on each output record. As you will see, the driver file controls when the data step is finished building the dataset.

Next, the income file is processed. Since DONE1 is initialized to false (see **[Line 2]**), "not DONE1" is true **[Line 6]**. The DONEx variables are used to signal when you are done processing a support file. The ADVANCE flag is initialized to false **[Line 7]**. The ADVANCE flag allows you to control when the file pointer is moved to the next record in the file. It is initialized to false because you never want the file pointer to advance on the first iteration through the DO loop.

The DO loop is entered since initially KEY1 is missing and LAST1 is not set to true **[Line 9]**. KEY1 (=2) is read in, holding it with a double trailing @@. Since the value of KEY1 (=2) is greater than CUSTNUM (=1) the DO loop is terminated **[Line 9]**. The values of KEY1 and CUSTNUM are then compared **[Line 14]**. If a match is found, the rest of the income information is read in **[Line 15]**. In this case, since no match is found, income is missing. Because the input line was held with a double trailing @@, the income file pointer is retained on this record for the next data step iteration. If you did not hold the line this way, you would lose the income information for this line.

The sales file is processed next. DONE2 is initially false (see **[Line 2]**) so the **[Line 18]** condition is true. Since KEY2 and LAST2 are initially missing, the DO loop is entered **[Line 21]**, and

KEY2 (=1) is read in **[Line 23]**. Since KEY2 (=1) is equal to CUSTNUM (=1) the DO loop is terminated **[Line 21]** and the subsequent check allows the remaining sales information to be input (SALES=20) **[Line 26-27]** which releases the current line.

Iteration 1 output data:

```
Custnum  Address  Income  Sales
1        Main       .      20
```

Iteration 2:
Initial file pointer positions:

```
   Driver        Income        Sales
   1 Main       -> 2 100         1 20
-> 2 First         3 60       -> 3 40
   5 Second        4 140          4 100
                   5 240
```

Again the driver file information is read in (CUSTNUM=2, ADDRESS=First) **[Line 4-5]**.

The income file is next. DONE1 is not set to true **[Line 6]**. LAST1 is not set to true and since the key values are retained (see **[Line 2]**) CUSTNUM (=2) is now equal to KEY1 (=2) so the DO loop is never entered **[Line 9]**. Notice that the file pointer has not changed from the first iteration because you used the double trailing @@. The income data is read in (INCOME=100) **[Line 14-15]**.

Moving on to the sales file, DONE2 is not set to true **[Line 18]**. At this point KEY2 has been retained at a value of 1. The DO loop is entered because KEY2 (=1) is less than CUSTNUM (=2) and LAST2 is not set to true **[Line 21]**. The second sales record is read in (=3) **[Line 23]**. The DO loop is terminated because KEY2 (=3) is now greater than CUSTNUM (=2). The associated sales information is never read in and is missing.

Iteration 2 output data:

```
Custnum  Address  Income  Sales
2        First     100     .
```

Iteration 3:
Initial file pointer positions:

```
   Driver        Income        Sales
   1 Main         2 100         1 20
   2 First     -> 3 60       -> 3 40
-> 5 Second       4 140          4 100
                  5 240
```

The last driver file record is read in (CUSTNUM=5, ADDRESS=Second) **[Line 4-5]**.

Again, proceeding to the income file first, DONE1 is not set to true **[Line 6]**. Since KEY1 has been retained at the value of 2 and LAST1 is not set to true, the DO loop is entered **[Line 9]** and the next income record is read in (=3) **[Line 11]**. The ADVANCE flag is set to true **[Line 12]**. KEY1 (=3) is still less than CUSTNUM (=5), so the DO loop does not terminate. The INPUT was held with a double trailing @@, so the file pointer remains on the current line. This is where the ADVANCE variable comes into play. ADVANCE is used as a flag that handles input buffer clearing. Since ADVANCE=true on all subsequent iterations of the DO loop, a single INPUT statement is executed that clears the previous INPUT @@ statement and advances the file pointer to the next record **[Line 10]**. The INPUT statement here is not used in a conventional sense, but only as a method of releasing the current line and advancing the file pointer to the next record:

```
  Income
   2 100
   3 60
-> 4 140
   5 240
```

The next income record is read in (=4) **[Line11]**. Since KEY1 (=4) is still less than CUSTNUM (=5), the loop is repeated.

```
  Income
   2 100
   3 60
   4 140
-> 5 240
```

The file pointer is advance **[Line 10]** and the next KEY1 value is read in (=5) **[Line 11]**. Now the DO loop is terminated because both criteria are met (the last income record has been read in and KEY1 >= CUSTNUM). Since now KEY1 (=5) and CUSTNUM (=5) are equal, the subsequent comparison **[Line14]** allows the reading in of the remaining income data (INCOME=240) **[Line15]**. DONE1 is set to true **[Line 16]** indicating that the processing on this file is completed.

Finally, the sales file is processed. DONE2 is not set to true. Since KEY2 has been retained at the value of 3 and LAST2 is not set to true the DO loop is entered and KEY2 (=3) is re-read **[Line 23]** (remember that this record is being held with a double trailing @@). ADVANCE is set to true **[Line 24]**. Since KEY2 (=3) is less than CUSTNUM (=5) the DO loop is repeated. The file pointer is advanced to the next record **[Line 22]** and the next KEY2 value is read in (=4) **[Line 23]**.

```
   Sales
   1 20
   3 40
-> 4 100
```

Because LAST2 is set to true, the DO loop terminates even though KEY2 (=4) is still less than CUSTNUM (=5). Since KEY2 and CUSTNUM do not match, the rest of the sales information is not read in and is missing. LAST2 is set to true and KEY2 is <= CUSTNUM therefore DONE2 is set to true **[Line 28]**. The processing on this file is completed.

Iteration 3 output data:

```
Custnum  Address  Income  Sales
5        Second     240     .
```

On what would be the next iteration of the data step, SAS tries to read past the end of the driver file and the data step is terminated. Since you do not prevent SAS from trying to read past the end of file only on the driver file, it is the driver file that dictates when the data step ends, rather than any of the support files.

The final merged SAS dataset contains the following information:

```
Custnum  Address  Income  Sales
1        Main       .       20
2        First      100     .
5        Second     240     .
```

Conclusion

1. Merging flat files directly allows you to bypass the creation of intermediate SAS datasets resulting in significant time savings.
2. The flat files must be merged in some common key order and there must not be any duplicate key values.
3. The use of the double trailing @@ allows re-reading a record over multiple data step iterations.
4. There is more coding effort with the flat file merge. But not that much greater. In both cases you need to specify the INPUT statements. The gains in processing time can more than make up for the added coding time especially when dealing with multiple large flat files that need to be processed in a production type environment (e.g. on a periodic basis).
5. The code above requires that you specify one of the input files as a driver file. However, this is not a strict requirement. You could easily modify the code to enable merging based on the shortest or longest file (number of observations). Also, the support file code could easily be made into a macro.

Contact Information

Robert Patten
Lands' End
5 Lands End Way
Dodgeville, WI 53533
608-935-4832
Robert.Patten@landsend.com