Paper 082-2011

Efficient Merging: Creating Format Libraries to Save CPU Time

Tyler Benz, Northrop Grumman, Monterey, CA

ABSTRACT

In my line of work, we are constantly reading in and writing out files containing millions of records and dozens of variables. It is not uncommon to receive a file containing a handful of records and be asked to merge it with multiple large files, do some data manipulation and write out the data to a new file. Programs like these tend to take a lot of CPU time and resources to run, so we are constantly looking for better ways to speed up processing time by implementing efficient SAS techniques within our code. PROC FORMAT and its user defined formatting options is one tool that can be used to do just that: SAVE TIME.

INTRODUCTION

Besides being a handy formatting tool in general, PROC FORMAT also boasts the ability to change a format into a dataset and vice versa with the CNTLIN/CNTLOUT options. It is with this CNTLIN option (and a couple of steps along the way) that will help us to create a "format library" from an existing dataset. The format library can then be used as a searching utility when reading in subsequent datasets. This paper will present the way to create a format library from an existing dataset in conjunction with PROC FORMAT that will cut processing time down substantially when merging large datasets on a MAINFRAME. I will compare this method of merging large datasets with three other methods: A simple merge (not containing the format library), using a macro in combination with a simple merge, and a Proc SQL join. My intent is to look at the CPU time taken for each of these techniques and determine the most efficient method of merging large datasets. These programs are all run on a MAINFRAME.

Suppose that we are given a file that contains 50,000 unique records with 20 variables (**FILE 1**). We are then asked to merge this file to three large **monthly** files (**FILE 2 (July), FILE 3(August), FILE 4(September)**) each containing 15 common payment variables. We only want to keep records that match with FILE 1. We are to also sum up the payment variables over the three months for each record and then write this new dataset out to a file. The final dataset will contain the variables from **FILE 1** as well as those from **FILE 2, FILE 3 and FILE 4** (which have the same variables) for records that match between the files.

There are many ways to go about programming the above scenario in SAS. That's the beauty of SAS – it's a dynamic tool that can be utilized in many ways to reach the same conclusions. So while I am sure you can think of other ways to gather the data, I have simply chosen a couple of ways that I have seen firsthand. **Figure 1** shows a diagram of how SAS will be programmed for **Method 1, Method 3 and Method 4**, and **Figure 2** shows a diagram of the SAS programming for **Method 2**, seen in the **INDEX**.

METHOD 1: SIMPLE MERGE - SET MONTHLY FILES, SUM AND THEN MERGE

This method will look identical to **Method 4**, except for the additional "format library" coding that will be added into it. The first step is to read in the data from **FILE 1**:

```
DATA FILE1;
INFILE FILE1;
INPUT
@001 SSN         $CHAR9.
@010 SERVICE     $CHAR1.
   …
@068 COMP        $CHAR1.

NOTE: 50000 records were read from the infile FILE1.
NOTE: The data set WORK.FILE1 has 50000 observations and 20 variables.
NOTE: The DATA statement used 0.98 CPU seconds and 16940K.
```

Next, I read in the three monthly files **(FILE 2, FILE 3, and FILE 4)** all at once so that they will be written to one dataset. I also **write an array** to change any pay values that are "-99999" or "-999" to 0:

```
DATA ACTIVE_DUTY;
INFILE ACT;

INPUT @057 PGRADE   $CHAR1. @;
IF PGRADE IN ('W','O') THEN DO;
```

```
INPUT @116 OPAY1    PD3.
      @131 OPAY2    PD3.
     …;
ELSE IF PGRADE IN ('E') THEN DO;

INPUT @116 EPAY1    PD3.
      @121 EPAY3    PD3.
     …;

ARRAY NVAR{*} _NUMERIC_;
DO I=1 TO DIM(NVAR);
   IF NVAR{I} IN (-99999,-999) THEN NVAR{I}=0;
END;

NOTE: 1616594 records were read from the infile ACT.
NOTE: 1613533 records were read from the infile ACT.
NOTE: 1613278 records were read from the infile ACT.
NOTE: The data set WORK.ACTIVE_DUTY has 4843405 observations and 17 variables.
NOTE: The DATA statement used 215.28 CPU seconds and 22362K.
```

Now that the three monthly files are combined into one dataset, we can **sum the pay variables by SSN** so that we have one unique record per person in the dataset with three months' pay summed up.

```
PROC SUMMARY DATA = ACTIVE_DUTY NWAY MISSING;
CLASS SSN;
VAR BASIC_PAY
    OPAY1
    OPAY2
    OPAY3
    …
    EPAY3
    EPAY4
    EPAY5;
OUTPUT OUT = SUMMED_PAY SUM=;

NOTE: There were 4843405 observations read from the data set WORK.ACTIVE_DUTY.
NOTE: The data set WORK.SUMMED_PAY has 1655235 observations and 17 variables.
NOTE: The PROCEDURE SUMMARY used 241.05 CPU seconds and 22406K.
```

Now we merge **FILE 1** with the combined, summed dataset containing **FILE 2, FILE 3 and FILE 4**, only keeping records that are found in both datasets.

```
DATA COMBINE;
MERGE FILE1 (IN=A) SUMMED_PAY (IN=B);
BY SSN;
IF A AND B;

NOTE: There were 50000 observations read from the data set WORK.FILE1.
NOTE: There were 1655235 observations read from the data set WORK.SUMMED_PAY.
NOTE: The data set WORK.COMBINE has 50000 observations and 37 variables.
NOTE: The DATA statement used 16.86 CPU seconds and 22863K.
```

The final step is to write the data out to a file.

```
DATA _NULL_;
SET COMBINE;
FILE OUTT;
PUT
@001 SSN          $CHAR9.
@010 SERVICE      $CHAR1.
@011 SEX          $CHAR1.
…
@135 EPAY4             4.
@139 EPAY5             6.
@145 BASIC_PAY         6.
```

```
;

NOTE: 50000 records were written to the file OUTT.
NOTE: There were 50000 observations read from the data set WORK.COMBINE.
NOTE: The DATA statement used 2.86 CPU seconds and 22607K.
```

When we include the other SAS code in the program but not seen above (Proc Prints, Freqs, Sorts etc.) the total CPU time totals **CPU=8:28.29**. That's 8 minutes and 28 seconds of CPU time – this can equate to hours of clock time, especially when there are many processes running on a mainframe at once.

METHOD 2: SIMPLE MERGE: MACRO AND MERGE COMBO

This method takes a different approach than the method we just saw above; however, the first part of the program stays the same (read in **FILE 1**). After reading in **FILE 1**, the next step is to set up a basic macro that will be used to **read in each monthly file separately** instead of all at once. I will also **merge each monthly file to FILE 1** separately within the macro as well.

```
%MACRO SASGF (I);

DATA ACTIVE_DUTY&I;
INFILE ACT&I;

INPUT @057 PG    $CHAR1. @;
IF PG IN ('W','O') THEN DO;

INPUT @116 OPAY1     PD3.
      @131 OPAY2     PD3.
             …
INPUT @110 BASIC_PAY PD3.
      @029 SSN    $CHAR9.;

ARRAY NVAR{*} _NUMERIC_;
DO I=1 TO DIM(NVAR);
   IF NVAR{I} IN (-99999,-999) THEN NVAR{I}=0;
END;

PROC SORT DATA = ACTIVE_DUTY&I; BY SSN;

DATA COMBINE&I;
MERGE FILE1 (IN=A) ACTIVE_DUTY&I (IN=B);
BY SSN;
IF A AND B;

%MEND;

%SASGF(1);
%SASGF(2);
%SASGF(3);


NOTE: 1616594 records were read from the infile ACT1.
NOTE: The data set WORK.ACTIVE_DUTY1 has 1616594 observations and 17 variables.
NOTE: The DATA statement used 71.41 CPU seconds and 22618K.

NOTE: There were 50000 observations read from the data set WORK.FILE1.
NOTE: There were 1616594 observations read from the data set WORK.ACTIVE_DUTY1.
NOTE: The data set WORK.COMBINE1 has 50003 observations and 36 variables.
NOTE: The DATA statement used 15.75 CPU seconds and 22582K.
```

… (Output above for **FILE 2**, similar output for **FILE 3 and FILE 4**)

This will save CPU time by eliminating unnecessary records from the PROC SUMMARY used to sum the pay variables in the next step. We implement the macro for the three large monthly files and then set them as one dataset before summing the pay variables. This time we also have to include all of the variables from FILE1 as ID variables to bring them into the new summed dataset.

```
DATA ACTIVE_DUTY;
SET COMBINE1 COMBINE2 COMBINE3;

NOTE: There were 50003 observations read from the data set WORK.COMBINE1.
NOTE: There were 49346 observations read from the data set WORK.COMBINE2.
NOTE: There were 48776 observations read from the data set WORK.COMBINE3.
NOTE: The data set WORK.ACTIVE_DUTY has 148125 observations and 36 variables.
NOTE: The DATA statement used 0.70 CPU seconds and 22710K.

PROC SUMMARY DATA = ACTIVE_DUTY NWAY MISSING;
CLASS SSN;
ID  SERVICE
    SEX
    …
    COMP;
VAR BASIC_PAY
    OPAY1
    OPAY2
    …
    EPAY5;
OUTPUT OUT = FINAL SUM=;

NOTE: There were 148125 observations read from the data set WORK.ACTIVE_DUTY.
NOTE: The data set WORK.FINAL has 50000 observations and 36 variables.
NOTE: The PROCEDURE SUMMARY used 5.31 CPU seconds and 22607K.
```

And we finish by writing the dataset to an output file like the previous method. Overall, the CPU Time used for this program is **CPU=5:22.59**. This is a vast improvement from **Method 1** – the biggest time saver for this program was merging the files with FILE 1 BEFORE summing the pay variables, because then only relevant records had their pay variables summed.

METHOD 3: PROC SQL

For this method using PROC SQL, nothing changes in the first couple of steps from **Method 1**. We read in **FILE 1**, and then we read in **FILE 2, FILE 3 and FILE 4** as one combined dataset. So at this point we have two datasets to work with. This is where the PROC SQL comes in. In the first part of PROC SQL, we are creating a table called '**SUMMED**' that will take any duplicate records, sum up each of the pay variables, and keep just one record per unique SSN. The second part of the PROC SQL creates another table called '**MERGED**', selects all variables from **FILE 1 and SUMMED** and keeps all matching records.

```
PROC SQL;
CREATE TABLE SUMMED AS
SELECT DISTINCT(SSN),
SUM(OPAY1) AS OPAY1_SUM,
SUM(OPAY2) AS OPAY2_SUM,
…
SUM(EPAY4) AS EPAY4_SUM,
SUM(EPAY5) AS EPAY5_SUM,
SUM(BASIC_PAY) AS BASIC_PAY_SUM
FROM ACTIVE_DUTY
GROUP BY SSN;
CREATE TABLE MERGED AS
SELECT *
FROM FILE1 AS A INNER JOIN SUMMED AS B
ON A.SSN=B.SSN;
```

The last step is to write the **MERGED** dataset out to a file. PROC SQL does multiple steps all at once, eliminating the need to sort the datasets when merging and can sum the pay variables as well. While it is might seem like this method would cut down CPU Time, our total for this program is **CPU=7:30.16**. Better than a simple merge but not nearly as good as using a macro and merging one monthly file at a time.

METHOD 4: FORMAT LIBRARY

The SAS program for this method is almost identical to **Method 1**. The key is to create a user defined format library from **FILE 1** used to then search the three monthly files for specific records as they are being read in.

The following is required to successfully create a format library:

- **There must not be any duplicates** with the variable that you are using as the **KEY** variable.
  - The **KEY** variable is what we will be searching for in the three monthly datasets (in our case this is SSN).
- **At least** these variables must be included in the dataset that you are making into a format library:
  - **FMTNAME**: The name of the format you are creating – can be whatever you would like besides an already defined format name.
    - When your KEY is character, FMTNAME must start with a **$** (just like any PROC FORMAT value)
  - **TYPE**: Specifies character ('C') or numeric ('N') format.
  - **START**: This is the value you want to format into a label (your KEY)
    - If you are specifying a range, START specifies the lower end of the range and END specifies the upper end.
  - **LABEL**: the label you are giving to your KEY variable.
    - Like any format, this is what your KEY variable will be formatted to when it is written to the format library. I generally use 'Y', but it can be anything. (Note – Make sure the label isn't a character that could be the first byte in your KEY)

Let's see **FORMAT LIBRARY** in action.

Like before, we read in **FILE 1**. But we now add four lines of code in the data step (the above requirements). We also need to make sure that there are no duplicate SSNs in our dataset, so we use the **NODUPKEY** option in PROC SORT. If you are sure that the file you are creating the format library with has no duplicates then you can skip using PROC SORT, though it never hurts to check and make sure that each record is unique.

```
DATA FILE1;
INFILE FILE1;
INPUT
@001 SSN        $CHAR9.
@010 SERVICE    $CHAR1.
   …
@068 COMP       $CHAR1.

START=SSN;
FMTNAME='$FSSN';
LABEL='Y';
TYPE='C';

NOTE: 50000 records were read from the infile FILE1.
NOTE: The data set WORK.FILE1 has 50000 observations and 24 variables.

PROC SORT DATA=FILE1 NODUPKEY;
BY SSN;

NOTE: 0 observations with duplicate key values were deleted.
NOTE: There were 50000 observations read from the data set WORK.FILE1.
NOTE: The data set WORK.FILE1 has 50000 observations and 24 variables.
NOTE: The PROCEDURE SORT used 0.89 CPU seconds and 17054K.
```

Now we use PROC FORMAT with the CNTLIN option to create our dataset into a Format Library. This user defined format is temporarily stored for this SAS session and will be deleted upon exiting.

```
PROC FORMAT CNTLIN=FILE1;
NOTE: Format $FSSN has been output.
```

The only other difference between this code and **Method 1** is two lines of code, implemented after the infile statement when reading in the three monthly files, seen in bold below.

```
DATA ACTIVE_DUTY;
INFILE ACT;

INPUT @029 SSN $CHAR9.@;
IF PUT(SSN,$FSSN.)='Y';

INPUT @057 PGRADE  $CHAR1.@;
IF PGRADE IN ('W','O') THEN DO;
```

```
INPUT @116 OPAY1     PD3.
      @131 OPAY2     PD3.
      …;

ARRAY NVAR{*} _NUMERIC_;
DO I=1 TO DIM(NVAR);
   IF NVAR{I} IN (-99999,-999) THEN NVAR{I}=0;
END;

NOTE: 1616594 records were read from the infile ACT.
NOTE: 1613533 records were read from the infile ACT.
NOTE: 1613278 records were read from the infile ACT.
NOTE: The data set WORK.ACTIVE_DUTY has 148125 observations and 17 variables.
NOTE: The DATA statement used 69.95 CPU seconds and 24442K.
```

Instead of our population including **all** members from the three monthly files, it contains just the records that matched our KEY variable – SSN. Though the format library isn't an actual merge, in that we still need to merge FILE 1 with the above dataset to combine all of the variables into one dataset, we have still cut down our population drastically. This will allow us to save CPU Time on any other data manipulation we may need to do throughout the rest of the program (in our case, PROC SUMMARY as well as the merge).

```
PROC SUMMARY DATA = ACTIVE_DUTY NWAY MISSING;
CLASS SSN;
VAR BASIC_PAY
    OPAY1
    OPAY2
    …
    EPAY5;
OUTPUT OUT = SUMMED_PAY SUM=;

NOTE: There were 148125 observations read from the data set WORK.ACTIVE_DUTY.
NOTE: The data set WORK.SUMMED_PAY has 50000 observations and 17 variables.
NOTE: The PROCEDURE SUMMARY used 3.72 CPU seconds and 22438K.

DATA COMBINE;
MERGE FILE1 (IN=A) SUMMED_PAY (IN=B);
BY SSN;
IF A AND B;

NOTE: There were 50000 observations read from the data set WORK.FILE1.
NOTE: There were 50000 observations read from the data set WORK.SUMMED_PAY.
NOTE: The data set WORK.COMBINE has 50000 observations and 41 variables.
NOTE: The DATA statement used 0.77 CPU seconds and 22895K.
```

After writing the data out to a file, the total CPU Time for the program is **CPU=1:28.90**.

This is by far the best CPU Time that we have seen. By adding just a few lines of code, we have effectively cut down on the time it took to capture the exact same population as the other methods.

CONCLUSION

To ensure that the four final datasets created contained the exact same data, I ran each file against one another using PROC COMPARE.

```
Number of Observations in Common: 50000.
Total Number of Observations Read from WORK.FORMAT_LIB: 50000.
Total Number of Observations Read from WORK.SIMPLE_MERGE: 50000.

Number of Observations with Some Compared Variables Unequal: 0.
Number of Observations with All Compared Variables Equal: 50000.

NOTE: No unequal values were found. All values compared are exactly equal.

*****
Number of Observations in Common: 50000.
Total Number of Observations Read from WORK.SIMPLE_MERGE: 50000.
```

```
Total Number of Observations Read from WORK.MACRO_MERGE: 50000.

Number of Observations with Some Compared Variables Unequal: 0.
Number of Observations with All Compared Variables Equal: 50000.

NOTE: No unequal values were found. All values compared are exactly equal.

*****
Number of Observations in Common: 50000.
Total Number of Observations Read from WORK.MACRO_MERGE: 50000.
Total Number of Observations Read from WORK.PROC_SQL: 50000.

Number of Observations with Some Compared Variables Unequal: 0.
Number of Observations with All Compared Variables Equal: 50000.

NOTE: No unequal values were found. All values compared are exactly equal.
```

We can see that the methods were able to create identical results - The differences can only be seen in the CPU time taken for each method, which is a very important aspect for efficient programming.

The total CPU Time used for each method:

Method 1 (SIMPLE MERGE):	**8:28.29 CPU Minutes**
Method 2 (MACRO/MERGE):	**5:22.59 CPU Minutes**
Method 3 (PROC SQL):	**7:30.16 CPU Minutes**
Method 4(FORMAT LIBRARY):	**1:28.90 CPU Minutes***

To me, the main advantage of creating a format library is its simplicity – as long as you follow the necessary guidelines, you can successfully cut CPU Time down substantially in your programming. I am also confident that there are many more applications that can be used with this tool – what I have shown above is merely one way to use it. If you are able to implement format libraries in other efficient and helpful ways, please let me know! (Contact info at the end of this paper)

INDEX

Figure 1: Overview of the programming for Method 1, 3 & 4

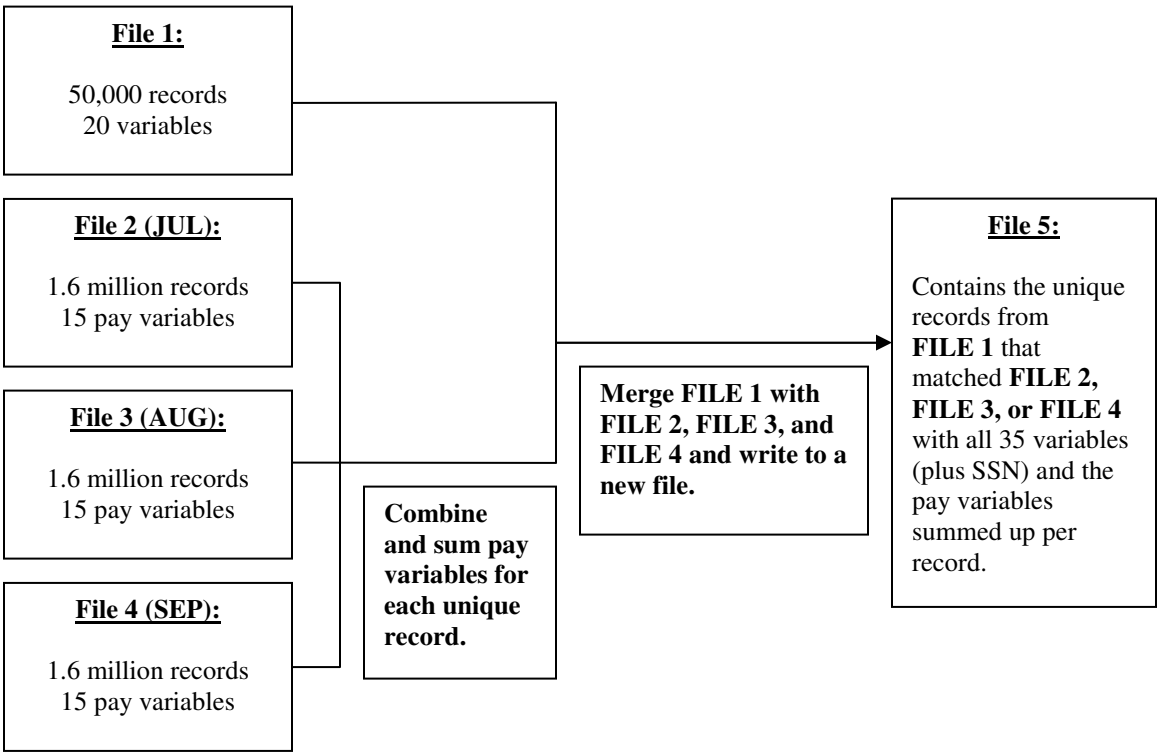
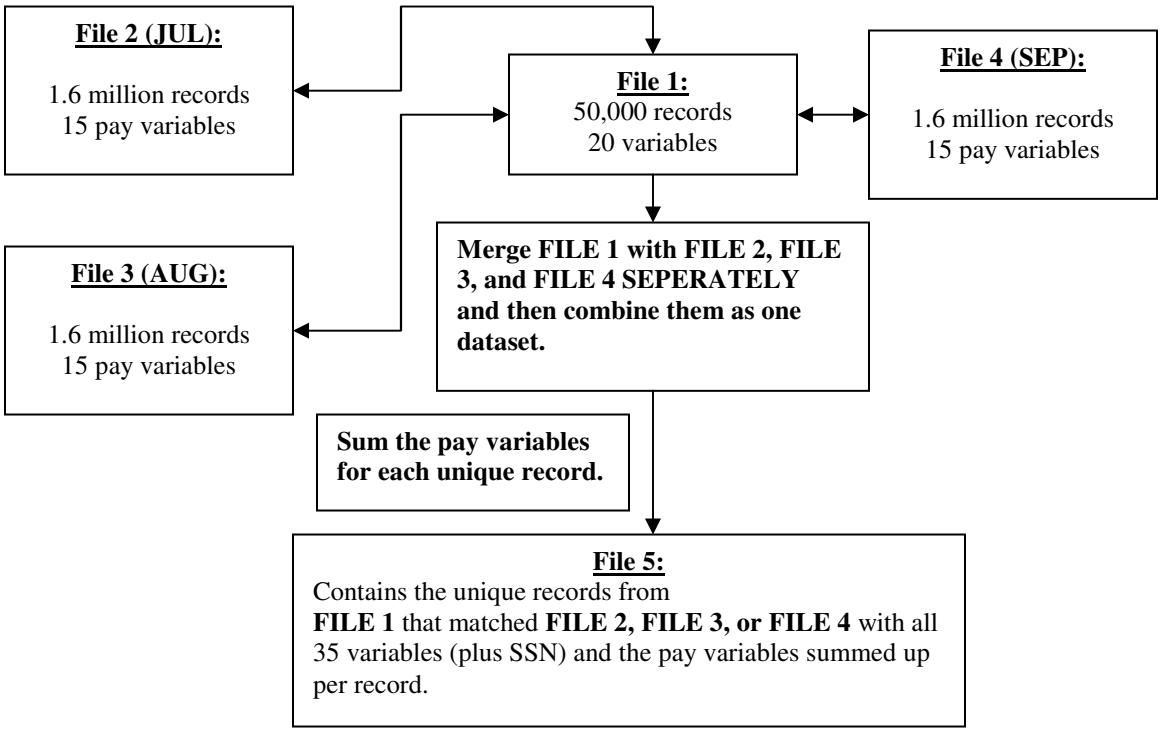


Figure 2: Overview of the programming for Method 2



REFERENCES

1. SAS Institute Inc., 2004. *SAS® 9.1 SQL Procedure User's Guide*. Cary, NC: SAS Institute Inc.
2. McGowan, K. and Spruell, B. "Proc SQL Tips and Techniques - How to get the most out of your queries". <http://analytics.ncsu.edu/sesug/2005/TU09_05.PDF>
3. Bilenas, J. "I Can Do That With PROC FORMAT", SAS Global Forum 2008, San Antonio, TX. <http://www2.sas.com/proceedings/forum2008/174-2008.pdf>

ACKNOWLEDGMENTS

I would like to thank all of the hard working analysts at the Defense Manpower Data Center for introducing me to such a powerful and useful tool.

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the author at:

Tyler Benz
Email: TylerBenz@Gmail.com