Paper BtB-01

5 Simple Steps to Improve the Performance of your Clinical Trials Table Programs using Base SAS® Software

Performance is defined in the Encarta® World English Dictionary as "**manner of functioning:** the manner in which something or somebody functions, operates, or behaves". In order to improve the performance of a SAS program, we need to look at several factors. These include not only the processing speed of the SAS program but also the readability and understandability of the program so that it can be edited efficiently.

Most clinical trials tables involve taking several SAS data sets, which can be rather large, combining them and then producing statistics ranging from simple counts and means, using PROC FREQ or PROC MEANS, to more complex statistical analysis. There are a few practices that each programmer can embrace to make these programs more efficient, thereby increasing the performance of these programs.

This paper will detail these steps, with examples, so that you can add them to your SAS tool bag for your next set of project outputs.

Pre-Step: Understand your baseline

Before you can start improving the efficiency of your SAS programs, you must first understand where you are – your baseline. Use the SAS option FULLSTIMER to reveal the most information about your current performance. This option will write real time (ie: elapsed or wall time), CPU time (user and system) and memory statistics to the SAS log at the end of each step and at the end of the program.

For example:

```
NOTE: The data set WORK.FINAL1 has 155 observations and 167 variables.
NOTE: DATA statement used (Total process time):
      real time           0.03 seconds
       user cpu time       0.01 seconds
       system cpu time     0.03 seconds
       Memory                            1490k
       OS Memory                         8668k
       Timestamp            7/5/2013  3:51:44 PM
```

Example 1: Output of FULLSTIMER option

Step 1: Reduce sorting

I/O is one of the biggest uses of resources and sorting your data, especially large data, can really impact your performance. There are several strategies for reducing sorting. The ones I like to use are:

- Group like tasks together: if you have several steps that require data in the same order, group those tasks together in your SAS program. This seems very logical but is often missed when following an output specification. A few minutes spent planning your program to take advantage of this can reap large rewards.

- Use an index: creating an index does take up resources but if you have programs that need the data sorted repeatedly using the same BY, it is usually better to create and store an index with the data than to resort the data several times. Use PROC DATASETS to create an index on an existing SAS data set:

```
proc datasets library=work;
modify SAS_data_set;
index create key_var;  → Simple Index
index create key=(byvar1 byvar2);  → Composite  Index
run;
```

Example 2: Creating An Index

- Use SQL: PROC SQL can be a good alternative to this sequence: SORT → SUBSET/MERGE → SORT → PRINT. These tasks can be accomplished in 1 SQL call. Here is a simple SQL that brings in data, subsets, sorts and prints the data.

```
proc sql;
select * from sashelp.class
where sex='F'
order by name;
quit;
```

Example 3: Use PROC SQL as an alternative

```
Name        Sex         Age         Height           Weight

Alice       F           13            56.5             84
Barbara     F           13            65.3             98
Carol       F           14            62.8          102.5
Jane        F           12            59.8           84.5
Janet       F           15            62.5          112.5
Joyce       F           11            51.3           50.5
Judy        F           14            64.3             90
Louise      F           12            56.3             77
Mary        F           15            66.5            112
```

Output 3: Output from SQL

Step 2: Bring in only what you need

A derived data set may contain hundreds of variables and many observations that you will not need in your output. The best approach here is to subset the data based on exactly what you need. Using a WHERE clause will subset the observations before the data is brought into the Input Buffer, while using an IF statement to subset the data means that all the data is brought into the Input Buffer sequentially. You can also use KEEP= to limit the variables to only what you need.

If you are really tight on resources, you may want to consider creating a few new variables and dropping the originals. Let's think about using flag variables. I'm sure we all have them in our derived data sets and normally many more than one or two per data set. They are usually defined as numeric with a values of 0,1 or <missing>. Each of these flag variables is 8 bytes. However, if they are changed to character variables, they can be reduced to 1 byte each – saving many precious bytes of space.

SAS Variable Type	Default length
1 stored as a SAS numeric	8 bytes
'1' stored as a SAS character	1 byte

Table 1: Comparison of Lengths

Step 3: Use procedures wisely

Consider how procedures work and use them to your best advantage. A good example is PROC FREQ, frequently used to produce counts for a table program. You could program your FREQ to use specific values passed in on a WHERE statement. We will call this "Method A".

For example:

```
      %macro freqit(dsn=,trt=);
         proc freq data=&dsn;
          tables lbtestn / out=tests&trt;
          where trtsort=&trt;
        run;
      %mend;
      %freqit(dsn=labs,trt=1)
      %freqit(dsn=labs,trt=2)
      %freqit(dsn=labs,trt=3)
```

Example 4: PROC FREQ example

This results in the procedure being called 3 times. The log, with resource usage data, is shown below:

```
83         %freqit(dsn=labs,trt=1)
MPRINT(FREQIT):   proc freq data=labs;
MPRINT(FREQIT):   tables lbtestn / out=tests1;
MPRINT(FREQIT):   where trtsort=1;
MPRINT(FREQIT):   run;

NOTE: There were 109407 observations read from the data set WORK.LABS.
      WHERE trtsort=1;







NOTE: The data set WORK.TESTS1 has 43 observations and 3 variables.
NOTE: The PROCEDURE FREQ printed pages 1-2.
NOTE: PROCEDURE FREQ used (Total process time):
      real time           0.98 seconds
      user cpu time       0.20 seconds
      system cpu time     0.68 seconds
      Memory                            1268k
      OS Memory                         7388k
      Timestamp            7/11/2013  10:28:46 AM
```

Output 4: PROC FREQ log file

```
84          %freqit(dsn=labs,trt=2)
MPRINT(FREQIT):   proc freq data=labs;
MPRINT(FREQIT):   tables lbtestn / out=tests2;
MPRINT(FREQIT):   where trtsort=2;
MPRINT(FREQIT):   run;

NOTE: There were 120054 observations read from the data set WORK.LABS.
      WHERE trtsort=2;
NOTE: The data set WORK.TESTS2 has 43 observations and 3 variables.
NOTE: The PROCEDURE FREQ printed pages 3-4.
NOTE: PROCEDURE FREQ used (Total process time):
      real time           0.88 seconds
      user cpu time       0.25 seconds
      system cpu time     0.62 seconds
      Memory                            349k
      OS Memory                         7388k
      Timestamp            7/11/2013  10:28:46 AM

85          %freqit(dsn=labs,trt=3)
MPRINT(FREQIT):   proc freq data=labs;
MPRINT(FREQIT):   tables lbtestn / out=tests3;
MPRINT(FREQIT):   where trtsort=3;
MPRINT(FREQIT):   run;

NOTE: There were 114515 observations read from the data set WORK.LABS.
      WHERE trtsort=3;
NOTE: The data set WORK.TESTS3 has 43 observations and 3 variables.
NOTE: The PROCEDURE FREQ printed pages 5-6.
NOTE: PROCEDURE FREQ used (Total process time):
      real time           0.87 seconds
      user cpu time       0.14 seconds
      system cpu time     0.68 seconds
      Memory                            349k
      OS Memory                         7388k
      Timestamp            7/11/2013  10:28:47 AM
```

Output 5: SAS Log -- Method A

Instead of passing in values on a WHERE statement, use that variable, in our case TRTSORT, as one of the analysis variables on the TABLES statement. We will call this “Method B”. For example:

```
proc freq data=&dsn;
    tables trtsort*lbtestn / out=tests;
    where trtsort in (1,2,3);
run;
```

The SAS log shows the resources used:

```
76         proc freq data=labs;
77             tables trtsort*lbtestn / out=tests;
78             where trtsort in (1,2,3);
79
80         run;

NOTE: There were 384574 observations read from the data set WORK.LABS.
NOTE: The data set WORK.TESTS has 172 observations and 4 variables.
NOTE: The PROCEDURE FREQ printed pages 1-4.
NOTE: PROCEDURE FREQ used (Total process time):
      real time           0.96 seconds
      user cpu time       0.24 seconds
      system cpu time     0.70 seconds
      Memory                            1485k
      OS Memory                         7396k
      Timestamp            7/11/2013  12:34:43 PM
```

Output 6: SAS Log -- Method B

And the savings were substantial! These numbers are based on our system with our data. You may see different results but you should definitely see resource improvement with this technique.

	Method A	Method B	Pct Improvement**
User CPU time	0.59 seconds	0.24 seconds	59.3%
System CPU time	1.98 seconds	0.70 seconds	64.6%
Memory	1966k	1485k	24.4%
OS Memory	22164k	7396k	66.6%

**Pct Improvement calculated as difference between Method A and Method B, divided by Method A.

You should also utilize the percent calculation of procedures like FREQ when using the number of non-missing subjects as your denominator. This prevents you from having to recreate these calculations. Make sure you are getting the most from each procedure call!

Step 4: Simplify!

As mentioned at the beginning of this paper, one factor in performance is the readability and understandability of the program so that it can be edited efficiently. We all know that client updates come frequently and it is so much better if the program is easy to understand and easy to update.

Take the time to review your SAS code from time to time to make sure that it is easy to understand. If the code requires complex logic and complex programming, add comments so the next programmer understands what that section of code does. It is very easy to spend several hours looking at someone else's code, adding PROC PRINTs and PUT statements, just to understand what they did. This can have a negative impact on the program performance from the standpoint of efficient updates/changes to the program.

Another area to simplify is in the use of SAS macros. Macros can be a wonderful addition to your SAS program. In certain cases, they can also be wasteful – especially if you have nested macro definitions. You should never nest macro definitions, which causes macros defined within the outer macro to be recompiled for every iteration of the outer macro. If you need nested macros, define them separately. You can then call one macro from another as needed.

From the viewpoint of complexity, having nested macro calls can impede the understanding of what is really getting pushed to the SAS processor. If you must use this logic, make sure that the code is well documented.

For example:

```
%macro inner;
  <SAS code here>;
%mend;

%macro outer;
  <SAS code here>;
  %inner;
%mend;

%outer;
```

is generally more efficient than this example:

```
%macro outer;
  <SAS code here>;
  %macro inner;
    <more SAS code here for INNER>;
  %mend;
  %inner;
%mend;
```

Example 6: Nested Macro Definitions

In table programs, it is common practice to have a GEN file that is used to create similar tables. These GEN files typically use macro variables to pass in the information needed for a specific table. Many people define their GEN program as a SAS macro, and then define other macros within it. This can over-complicate the process and reduce efficiency, as shown above.

For example:

```
%gen_table(pop=safety,dsn=data_set,w=,dis1=criteria1,
dis2=criteria2)
```

Example 7: Using a GEN file

An alternative approach is to define macro variables using %LET statements and then bring in the code in the GEN file via %INC. The GEN file, in this case, does not start with %MACRO.

```
%let pop=safety;
%let dsn=data_set;
%let w=;
%let dis1=criteria1;
%let dis2=criteria2;
%inc "&root.&progdir\gen_qc_table.sas";
```

Example 8: Using %INC with %LET statements

If you have code that assigns values using a series of if statements, consider using PROC FORMAT instead of IF statements. This method is generally more efficient, both in terms of resources and simplicity to update. For example:

```
proc format;
  value trt;
  1 = ‘Treatment A’
  2 = ‘Treatment B’
  3 = ‘Treatment C’
  4 = ‘Placebo’
  ;
data table;
  set SAS_data_set;
  treat=put(trt,trt.);
run;
```

Example 9: Using PROC FORMAT to assign values

If you have many of these assignments to make, you can define them all in a single PROC FORMAT, so the procedure is called only once.

Step 5: Comment code and review

The last of the 5 steps for improving efficiency is to comment your code and periodically review it. Always add enough comments so that the next programmer can quickly find what they need and make updates as required. As mentioned earlier, this is especially important when the logic is complex or you are using a less-common SAS statement or procedure. A good example of this is when PROC SQL is used to join data. If the next person is not well-versed in PROC SQL, they may not understand some of the logic used. A few comments before that code block can make a world of difference.

Another important step is to review the code. A typical table program may start out as simple and then becomes complex one update at a time, as written in the specification. After a few updates, it is usually beneficial to review the code to see if some of the code can be consolidated, use fewer sorts or a different SAS statement. As code is added to, it sometimes becomes less efficient just by the process of adding new sections. Before beginning a large update, or when you have time, it is beneficial to do a code walk-through to see if the program can be re-organized so that it performs better – whether in terms of system resources or readability by the next programmer.

Extras

Here are a few SAS options and tips that can be used when writing SAS programs. They can help in many ways as you work to improve your SAS code.

Options

ERRORABEND: This SAS option can help as you debug your program. If your code produces an error, this option tells SAS to stop executing the entire program. It will save you much real/wall time waiting for a job to end, only to find that you had an error and will need to rerun.

OBS=: You can also use the OBS= system option to limit the number of observations read or written in all SAS data sets within the program. This can be useful when testing logic and you want to speed up your test. You can also use the data set option OBS= if you only want to limit observations in one particular data set.

SORTEDBY=: This option sets the sort indicator, which SAS uses to determine whether or not to sort the data. You may also want to set SORTVALIDATE so that SAS will validate this sort order. Default value is NOSORTVALIDATE.

MSGLEVEL=I: This option will write out Information notes (INFO) to the SAS log. With this option, an INFO note is written to the SAS log when an index is used.

MPRINT/MLOGIC/SYMBOLGEN: These macro debugging options can help you understand what is actually being interpreted by the SAS compiler.

Tips

WHERE= vs. IF: Using the WHERE= data set option, or WHERE statement, is generally more efficient than a subsetting IF statement in a DATA step. The exception is when a SAS function appears on the WHERE clause – this causes SAS to read the entire data set sequentially. You can sometimes use the WHERE operators to mimic the SAS function. For example:

```
     where protocol =: 'XYZ';
```

would produce the same results as:

```
     if substr(protocol,1,3)='XYZ';
```

Create a hash object: A hash object is held in memory so the benefit for these is less I/O when combining data sets. A good application for hash tables is a table look-up where the table used for look-up is relatively smaller than the table receiving the look-up results.

Conclusion

As you have seen, there are many, many ways to improve efficiency in your SAS programs. If you search the Internet or review past SAS Global Forum and regional conferences proceedings, you can find even more tips. Whether your goal is to make your program run faster (wall time), use less I/O or be easier to update, I hope that you have found several tips that you can use.

Acknowledgments

The author would like to thank the following reviewers: Jan Squillace and Martha Messineo from SAS, Magdalena Lopez and Hardik Sheth from Quintiles.

Additional Reading

- McGowan, Kevin>. 2013. "Big Data, Fast Processing Speeds". *Proceedings of the SAS Global 2013 Conference*. Cary, NC: SAS. Available at http://support.sas.com/resources/papers/proceedings13/036-2013.pdf

- SAS® 9.3 Companion for Windows

- SAS 9.3 Language Reference. See SAS System Concepts → Optimizing System Performance. Available at http://support.sas.com/documentation/cdl/en/lrcon/65287/HTML/default/viewer.htm#p1xjhzwjv6ojukn18mi4j1ysye76.htm

Contact Information

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

Name:	Sally Walczak
Enterprise:	Quintiles
E-mail:	sally.walczak@quintiles.com