Getting More For Less: A Few SAS® Programming Efficiency Issues

Arthur L. Carpenter
California Occidental Consultants

ABSTRACT

The old saying goes “If there is one way to do it in SAS®, there are three ways to do it.” Inevitably, when this is said in a group, there will be someone (probably a Quality Partner™) who will chime in with “And I will show you a fourth way.” It is of course wonderful to be working with a language that has the flexibility of multiple programming solutions, but this flexibility can come with a price. Very often the most obvious coding solution turns out not to be the most efficient one. It is not unusual to find programs written by programmers with a view to the end result, but with no understanding of the process of the best way of getting there.

If there are multiple solutions to a programming problem, which solution is the correct one? And then the bigger question becomes, ‘Does it make a difference?’

KEYWORDS

Efficiency, programming style

INTRODUCTION

Just as there are many paths to the top of the proverbial mountain, most programming problems have more than one coding solution. Each solution has tradeoffs; a complex program may execute very fast but may be difficult to maintain, character variables of minimal length save on data set size but may minimize stored information, and a program designed for ease of use may require more resources for both development and maintenance. Efficiency in programming is much more than selecting the fastest or coolest statement. It is much more about your approach to the task of creating a programming solution to a problem.

As the programmer/developer you should ask some questions before you start to write code.

Ž Should the program be easy to maintain?
Ž Should it execute as fast as possible?
Ž Should the data base size be minimized?
Ž Do system resources such as memory need to be conserved?

The knee jerk response is ‘Yes’ for each of the above questions, but of course in actuality, it may not be possible to create a solution that is optimal in each area. For instance compressed data sets will minimize data set size, but compression also slows the processing of the program.

The solution is rarely simple. Consider the problem of writing a program to save time. One needs to ask what kind of time or perhaps whose time? Is time a measure of the time to process (CPU Time), time to read and write data (I/O Time), or time to create and maintain the program (Programmer Time)?

As you consider these issues, you will develop a programming approach. Your approach to the problem, and your choice of procedures and statements will determine the efficiency of your program.

WHAT IS EFFICIENCY? - THE PROS AND CONS

Given the conflicting issues, the definition of efficiency itself comes into question. An efficient program must not only solve the problem for which it is written, but should also meet the goals of the developer based on questions such as those expressed above. Based on these questions priorities must be dtermined, these priorities become program goals, and an efficient program will meet these goals.

Ž Should the program be easy to maintain?
Complex programs are more difficult to maintain. Very often increased complexity can decrease execution time. If the increased complexity saves a few seconds, but if maintenance time increases by hours, the tradeoff may not be worth it.

Ž Should it execute as fast as possible?
Normally faster is better, but the cost of development and maintenance may easily offset any savings.

Ž Should the data set size be minimized?
Smaller data sets require less disk storage space, and this is a worthy goal. If the decreased size is achieved by compression, however, additional time will be required to decompress and then recompress the data set whenever it is read or rewritten.

Decreased size can also be achieved by reducing the length of data set variables, however this may also result in a reduced ability to store accurate or complete information.

Ž Do system resources such as memory need to be conserved?
Macro variables can be retrieved quickly because they are stored in memory, however large numbers of macro variables will reduce available memory. Reduced available memory may further limit the use of statements such as the CLASS statement in PROC MEANS.

Inefficient programs are usually created by programmers that do not even ask these questions. These programs often lack a specific coding style or purpose.

PROGRAMMING STYLE

Your coding style is determined to a great extent on your determination of the priorities outlined in the previous section. The style of the program should reflect a thoughtful evaluation of all of theses issues. All too often milliseconds are gained at the expense of hours of programming time.

Many programmers take pride in writing code that is both quick and uses the ‘just right’ set of techniques. While this may demonstrate programming expertise, it may not truly be the best program style. Develop a programming style that takes into account the above issues while also utilizing the best appropriate set of program statements.

Once you have determined your programming priorities, you will need to plan your program to take advantage of the things that will make a difference.

THINGS THAT MAKE A DIFFERENCE

Ž Minimize steps
Of all the things that you can do to make a program more efficient, this one can have the greatest impact. Evaluate the structure of the program to minimize the number of individual steps. Combine DATA steps and PROC steps when possible. Usually this will mean that you will also make fewer passes of the individual data sets.

Ž Minimize the number of data set reads
When reading a data set do everything with it that you can as soon as you can. A classic example that I encountered used a series of 17 DATA steps to break a single large data set into 17 distinct subsets. This required 17 passes of the large data set. The programmer had been unaware that the DATA statement allows multiple data sets to be designated with output controlled with the OUTPUT statement.

Ž Minimize the use of PROC SORT
Sorting on any computer is always time consuming. Try to minimize the number of sorts required. Many data sets that are created by PROC steps (*e.g.* MEANS, SUMMARY, FREQ, REG) will be created in a known sort order. Anticipate the order and avoid the sort. When a data set must be sorted and resorted multiple times consider creating indexes or at least try to fully exploit each sort. Many operations and PROC steps that require sorting can be avoided by the use of SQL.

Ž Use table lookup techniques
A long series of IF statements can easily slow a DATA step. When making a series of mutually exclusive IF-THEN determinations use an ELSE to set up successive IF statements. This way once an expression is TRUE further IF statements will not be processed. Consider the following series of IF statements used to designate a part name from a part code :

```
if code=1 then part='abc';
if code=2 then part='def';
if code=3 then part='ghi';
```

Regardless of the value of CODE, all three IF statements are executed. Preceding all but the first IF with an ELSE, means that as soon as one of the expressions is true the remaining ELSE IF statements are not executed. The code becomes:

```
if code=1 then part='abc';
else if code=2 then part='def';
else if code=3 then part='ghi';
```

A further gain in efficiency can be realized by avoiding the use of IF-THEN-ELSE processing altogether. A format can be created by using PROC FORMAT and the PUT or INPUT functions.

```
proc format;
value coder
  1='abc'
  2='def'
  3='ghi';
run;
```

```
data new; set old;
  part = put(code,coder.);
  run;
```

Ž Write the LOG to a file
For programs that generate a large LOG, rerouting the LOG to a file by using either the ALTLOG= option or PROC PRINTTO, can save both time (some time is spent generating the video display of the LOG) and memory (the LOG is stored in memory, which on large jobs may require intervention to clear).

Ž Use the WHERE statement/option
The WHERE statement can be much more efficient than the subsetting IF. Also, unlike the subsetting IF, it can be applied directly to the incoming data set as a data set option (recommended method).

```
set old(where=(code<5));
```

When an index is present the WHERE will, if appropriate, use it to maximum benefit. The gains in efficiency over the subsetting IF can be substantial, because only the observations that meet the criteria are actually read from the incoming data set.

Since the WHERE can be used in most PROC steps, the use of DATA steps that only create a data subset that is to be used in a procedure (say PRINT), can be avoided. This reduces the number of steps and the number of data reads.

A BIT OF FINESSE
There are many things to think about when coding for efficiency. Because of space limitations, only a small fraction of these can be included in this paper. While the following items may not provide as great a gain as those noted above, they are no less important as tools, and they should definitely be understood by the programmer.

Ž Compression
Used to reduce storage requirements, compression can be used to reduce data sets by as much as 85%. The greatest gains will be for data sets with long character variables that contain repeated values (such as blanks). Because the data set must be compressed and then uncompressed prior to use, there is an increase in processing time. Compression is by default off and can be turned on globally by using the COMPRESS system option or individually for a specific data set by using the COMPRESS= data set option.

Ž Indexes
Indexes allow the programmer to set up one or more logical sorts on a data set without physically sorting it. This can reduce the number of physical sorts of a data set and can decrease data set access time when using a WHERE. The cost is in the creation and maintenance of the indexes. The indexes must be recreated each time the data set is modified and the indexes themselves can take up significant storage space.

Ž Nested functions
The nesting of function calls can reduce the number of variables that need to be created in a DATA step. The following statements retrieve the two characters following the '-' in the variable STATION.

```
col = index(station,'-');
depth = substr(station,col,2);
```

Creation of the variable COL can be avoided by nesting the INDEX function within the call to the SUBSTR function.

```
depth = substr(station,index(station,'-
'),2);
```

This small efficiency gain can be offset by more complicated and harder to maintain code. Nesting function calls more than three deep is generally thought to be excessive (from a code maintenance perspective).

Ž Compiled stored macros
When a macro is first called it must be compiled before it can be executed. The compiled code can be stored permanently so that it will not need to be compiled again in a later session. Individually the processing gain is usually small, however if an application has many macro calls using stored macros can result in significant decreases in processing times. Be sure to update the stored code each time the macro definition changes, otherwise the new macro definition will remain uncompiled and the old compiled version will continue to execute.

Ž Structured Query Language, SQL
PROC SQL is used to include SQL statements into traditional SAS programs. In a number of situations SQL can be used to create data sets that are both sorted (without using PROC SORT) or summarized (without using MEANS or SUMMARY). Often the use of SQL reduces the number of steps needed and may also reduce processing time. SQL is not always more efficient in processing time, in the number of statements, or in the complexity of the code.

Ž Use the KEEP= and DROP= data set options
Reduce the number of variables that must be carried along from data set to data set by eliminating unneeded variables as soon as possible. By using the KEEP= option, the programmer also documents the variables that are a part of a specific data set.

Ž Use the RETAIN statement to initialize values.
Because assignment statements are executed for each incoming observation, constant values should always be assigned through the RETAIN statement. Unlike the assignment statement the retain statement is applied during the compilation of the DATA step. Similarly variable initialization can be made through the ARRAY statement.

Ž Control variable length
Use the LENGTH statement to make sure that any given variable does not take up more storage space than is necessary. Numeric variables that serve as flags or codes can often be stored as character values with no loss of information. Integer values such as SAS dates can be stored precisely in four bytes rather than the default length for numeric variables of eight bytes.

Ž Data set structure
Since different types of analysis require different data set structures, plan your data set structure according to how the data are to be used. Try to avoid a succession of either DATA steps or the use of PROC TRANSPOSE to rearrange your data.

SUMMARY

Writing efficient code is not as simple as just making the program execute as fast as possible. You must take into consideration a number of other factors. Is faster as important as maintainable code? What resources are most important; processing time, storage space, memory, programmer time?

Select the objectives of the program and then select your coding style to reflect these objectives. It is not sufficient to say that you just want the code to be fast or cool. Although I must say, that sometimes, just writing cool code can be an objective in its own right.

REFERENCES

The easy to read and easy to understand reference manual for the topic of efficiency is *SAS® Programming Tips: A Guide to Efficient SAS Processing,* Cary, NC: SAS Institute Inc., 1990. 155pp. Although this text has been around for a long time it still is packed with good and relevant information.

ABOUT THE AUTHOR

Art Carpenter's publications list includes two chapters in *Reporting from the Field*, the two books *Quick Results with SAS/GRAPH® Software*, and *Carpenter's Complete Guide to the SAS® Macro Language* and over three dozen papers and posters presented at various user group conferences. Art has been using SAS since 1976 and has served as a steering committee chairperson of both the Southern California SAS User's Group, SoCalSUG, and the San Diego SAS Users Group, SANDS; a conference cochair of the Western Users of SAS Software regional conference, WUSS; and Section Chair at the SAS User's Group International conference, SUGI.

Art is a SAS Quality Partner™ and through California Occidental Consultants he teaches SAS courses and provides contract SAS programming support nationwide.

AUTHOR CONTACT

Art Carpenter
California Occidental Consultants
PO Box 6199
Oceanside, CA 92058-6199

(760) 945-0613

art@caloxy.com
www.caloxy.com

TRADEMARK INFORMATION