EFFICIENT TECHNIQUES FOR LARGE DATA FILES
Judith H. Mopsik, ORI, Inc.

In an average EDP organization, code is used for a period of ten years or more before it is discarded (Reference 1), and a small program that is run each day becomes as important to optimize as a large program that is run once a month. Today, as more and more programmers are using SAS to solve both large and small data management problems, it is important to learn how to use SAS efficiently. We are all familiar with why managers prefer SAS - SAS is easy to learn, easy to use, and helps get results to clients quickly. Because SAS is especially convenient to use on a very large data set, we must now address the major criticism of SAS - it can be inefficient and consequently expensive.

In this tutorial topic I will review some of the literature which addresses efficient SAS programming techniques for working with large data sets. I will then examine alternative ways of programming in SAS with large volumes of data and evaluating these techniques for efficiency.

REVIEW OF THE LITERATURE
Efficient SAS techniques for processing large data sets through the use of working files, random samples, sufficient statistics and careful planning were explored in detail by John Sail (Reference 2) several years ago. Additional techniques have been detailed in the SAS Applications Guide (1980 Edition). Using procedures that do not require sorted data, and saving the output of a procedure and using it in later DATA steps are detailed in both references with excellent examples. Both references emphasize working with just the variables you need to get the job done. Analysis of a random sample, a stratified random sample, or a direct access random sample can often give almost as good statistical precision for a dramatic reduction in cost.

Often, however, we are asked to do more than provide descriptive statistics on the data base. Several topics presented at the 1982 SAS Users Group International (SUGI) Conference evaluated SAS coding efficiency from a "non-statistical" point of view. Two SAS tutorials evaluated the efficiency of various table lookup techniques (Reference 3) and subset selection methods (Reference 4). A third tutorial explained, by example, procedures for conditionally executing DATA/PROC steps in the same job stream by the use of %INCLUDE (Reference 5). This illustrated the division "SAS way" of executing/avoiding data-dependent code.

A fourth tutorial presented at the SUGI 82 conference explored for efficiency several techniques for adding a constant to every observation in a data set (Reference 6).

In addition to the current SAS literature on this topic, all programming language manuals address the obvious, but often overlooked techniques for fine-tuning a program for more efficient performance. Regardless of the language (PLI, FORTRAN, COBOL, PASCAL, etc.) many of these rules apply: moving computations outside of a loop which need not be calculated within the loop; avoiding high dimensional arrays; avoiding mix-mode arithmetic; writing a list of conditional statements in the order in which the event is most likely to occur. Many languages allow the programmer to "turn off" unneeded features that use a lot of CPU time (e.g. checking for division by zero). The SAS user does not have this kind of control, but other opportunities for improving efficiency in SAS do exist.

Studies have shown that most of the running time in non-IO-bound programs is concentrated in 3% of the source text (Reference 1). This paper will identify areas in the DATA step where efficiency improvements are possible and will explain why different methods of achieving the same goal can vary in the amount of CPU time they use. It is important to keep in mind that the percentage of time saved is dependent on the content of the DATA step. The examples used to demonstrate alternate techniques are only examining specific instructions within a DATA step. Your savings will be different, depending on the exact nature of your DATA step. Savings noted in terms of "seconds" represent the amount of CPU savings for coding a specific instruction (or for using a specific approach) versus an alternative.

OVERVIEW OF SAS CODING CONSIDERATIONS
Several techniques are examined in this report for optimizing CPU execution time within the SAS DATA step. Savings can be identical for a 10,000 observation data set processed twice daily (60 times/month) and a 600,000 observation data set processed monthly. Data sets
with 25,000 to 100,000 observations (called "_LIMIT" in each example) were used in my analysis to demonstrate the CPU time consumed in a data step for:

-SAS Supervisor execution
-Use of the LINK
-Array processing
-Numeric/character conversion
-List/formatted input
-List/formatted output

Sample code and an analysis of the relative efficiency of each strategy is evaluated after each example.

SAS SUPERVISOR EXECUTION

In the two examples presented in Figure 1, a SAS data set was created and then read in subsequent data steps by the use of the SET statement. Saving intermediate data sets, and using work files (in these cases WORK.EXAMPLE is referenced by SET EXAMPLE) can be the key to fine-tuning any SAS program. In the first data step, after each execution of the DATA step, control is returned to the SAS Supervisor which reads another observation. In the second example (1.2), the data set is accessed by a SET statement inside of a DO WHILE loop, and control is not returned to the SAS Supervisor. The STOP statement in example 1.2 is necessary because the SET statement cannot fail (the usual way of ending a DATA step) since it is inside of a conditionally executed DO loop. DATA ONCE and DATA EACH will contain the same number of observations and variables, but DATA ONCE is created using only 25% of the CPU time required to create DATA EACH. Efficiency tests indicate that 14-16 seconds were used by the SAS Supervisor per million calls of the data step.

In order to assess any additional overhead that might be incurred, 10 in-line assignment statements were added to each data step (Figure 2). These results show that the cost of 10 assignments for each of a million observations is only one second. Any additional CPU time results when the SAS supervisor sets all variables to "missing" between each DATA step execution. This amounts to only .1 second/million initializations between DATA step executions.

In conclusion programmer control of the data is more efficient than SAS Supervisor control. (Refer to the tutorial titled The SAS Supervisor for an explanation of how the SAS Supervisor works).
LINK STATEMENT

A LINK statement tells SAS to go to another statement in the same DATA step and begin executing statements from that point. Execution continues until a RETURN statement is executed and control returns to the statement immediately following the LINK. This example (Figure 3) demonstrates the use of the SAS LINK statement. In every case, the LINK used twice as much CPU time as in-line execution of programming statements.

Adding some numerical computations to the LINK (Figure 4) further illustrates the CPU expense of the LINK. This example showed that the cost of the LINK was about 3 seconds/million executions. In some cases, of course, only a LINK can accomplish the results you need. In situations where you only need to use redundant code a MACRO will be more efficient than the LINK.

*EXECUTE THE DATA STEP;
DATA NOLINK;
  DO I = 1 TO __LIMIT;
  END;

*EXECUTE THE DATA STEP AND INVOKE LINK;
DATA LINKTO;
  DO I = 1 TO __LIMIT;
  LINK XX;
  END;
XX: RETURN;

FIGURE 3: LINK STATEMENT
NO COMPUTATIONS

*EXECUTE THE DATA STEP;
DATA NOLINK;
  DO I = 1 TO __LIMIT;
  RAND = UNIFORM(0);
  END;

*EXECUTE THE DATA STEP AND INVOKE LINK;
DATA LINKTO;
  DO I = 1 TO __LIMIT;
  LINK XX;
  RAND = UNIFORM(0);
  RETURN;

FIGURE 4: LINK STATEMENT
WITH NUMERIC CALCULATION

SAS ARRAY PROCESSING

When many variables are to be processed in the same way, the use of the SAS ARRAY feature is very convenient. In order to evaluate the efficiency associated with using ARRAYS two different applications were examined. Arrays of 5, 10 and 20 variables were compared for data sets with 25,000-100,000 observations.

The sample code for two table lookup procedures is shown in Figure 5. The results demonstrate that ARRAYS are more efficient than hard-coded IF-THEN-ELSE statements, and as the number of variables in the array (i.e., table size) increased, the ARRAY became even more preferable to the cumbersome use of the IF-THEN-ELSE procedure.

The sample code in Figure 6 demonstrates two ways of performing a repetitive process (multiplying every element in an array by the same factor).

The results indicated that, for repetitive operations, the use of ARRAYS was much less efficient than hardcoding those operations. Furthermore, the relative performance of the two did not seem to depend on the number of observations. However, the relative performance did depend on the number of variables (ARRAY size), with ARRAYS comparing less favorably as the number of variables increased. The "human" factor, of course, must be considered here. This approach might be taken as a last resort when the execution time of a production run could not be enhanced in any other way.

One must keep in mind that there is a lot of "overhead" when using ARRAYS. In traditional programming languages subscript range checking can be turned off after you have tested the program. The automatic subscript range checks performed by SAS are always in effect and this explains why using ARRAYS can be less efficient than hard-coded statements.
EXECUTING AN ASSIGNMENT FROM AN ARRAY
DATA EXAMPLE:
ARRAY _Y(1) Y1-Y20;
setter DATASET;
*Compute l;
X = _Y;

EXECUTING A SEQUENCE OF IF-THEN-ELSE STATEMENTS:
DATA EXAMPLE:
setter DATASET;
IF I = 1 then X = Y1;
ELSE IF I = 2 THEN X = Y2;
ELSE IF I = 20 THEN X = Y20;

FIGURE 5: ARRAY PROCESSING FOR TABLE LOOKUP

APPLY A FACTOR TO EACH ARRAY ELEMENT
DATA EXAMPLE:
ARRAY _Y(1) Y1-Y20;
ARRAY _X(1) X1-X20;
setter DATASET;
DO I = 1 TO 20;
   _X = FACTOR * _Y;
END;

FIGURE 6: ARRAY PROCESSING FOR REPETITIVE PROCESSES

APPLY A FACTOR TO EACH ARRAY ELEMENT
DATA EXAMPLE:
ARRAY _Y(1) Y1-Y20;
ARRAY _X(1) X1-X20;
setter DATASET;
DO I = 1 TO 20;
   _X = FACTOR * _Y;
END;

CHARACTER/NUMERIC CONVERSION
SAS automatically converts character variables to numeric variables (and vice-versa) according to the rules described on page 154 in the 1982 SAS Basics Guide. A simple example of when character to numeric conversion takes place is if a number is defined as a character variable in a previous DATA step and is later needed for a numeric calculation. Figures 7 and 8 show the code used to examine the CPU time required for SAS to convert numeric and character data. Efficiency tests demonstrated that the use of the INPUT function, for character to numeric conversion which also avoids the "NOTE: CHARACTER VALUES HAVE BEEN CONVERTED TO NUMERIC" in the LOG, was slightly more time-consuming than the internal SAS conversion. On the other hand, the use of the PUT function for numeric to character conversion, represents a considerable savings (almost 60%) over the automatic SAS execution for numeric to character conversion.

Both cases illustrate that the major efficiency problem here (as in all programming languages) is conversion. Specific declarations save time and avoid truncation and conversion problems. Managing your data with care, and converting before "mixing" modes in an arithmetic operation can greatly improve efficiency within a DATA step. Of course, if you can avoid converting your data (i.e., making sure that it is originally created according to its use in your program) you improve your efficiency by several orders of magnitude.

FIGURE 7: CHARACTER TO NUMERIC CONVERSION
**FIGURE 8: NUMERIC TO CHARACTER CONVERSION**

LIST, COLUMN AND FORMATTED INPUT

The INPUT statement describes how you want SAS to read values into variables from input data lines. The INPUT statement can be used in three basic ways:

- List input or format-free input
- Column input
- Formatted input

The sample programs in Figure 9 read character and numeric data using various formatting conventions available in SAS.

The overall results indicated that reading character data was more efficient (15 seconds/million variables x obs) than reading numeric data. However, for both character and numeric variables, the more detailed formatting resulted in considerable savings in CPU time. For example, the "$CHAR" format was 3.5 seconds faster/million variables x obs) than the "$" format.
LIST COLUMN AND FORMATTED OUTPUT

The PUT statement is used to write output lines. The form of the PUT is identical to the INPUT statement. The examples presented in Figure 10 use various format options with the PUT statement to print numeric data. The results again indicated that the more detailed the formatting, the more efficient the program was. List output (PUT X) used 15% more CPU time than formatted output (PUT 10.6); while column output (PUT 1-10 6;) used only 3% more CPU time than formatted output.

CONCLUSIONS

The results indicate that great care should be taken when fine-tuning SAS programs for performance efficiency. Optimization of "human" resources can be achieved by planning ahead for sufficient computer resources (time, quantity of output, adequate memory), saving intermediate files so you don't have to restart a job, and debugging the code on small samples of data.

SAS program optimization, on the other hand, can be achieved when the programmer is aware of how SAS consumes CPU time. The SAS Supervisor execution, array processing, and data conversion examples demonstrate how SAS can be used effectively with large data sets. However, ease of program maintenance and modification must also be factored into any efficiency formula (e.g., reducing redundant code by the use of ARRAYS or the LINK statement). The examples presented here demonstrate some alternative strategies that are available to the SAS programmer.

*OUTPUT "REAL" DATA IN A LIST;
 DATA LIST;
 DO I = 1 TO _LIMIT;
   X = UNIFORM(0);
   FILE DD;
   PUT X;
 END;

*OUTPUT "REAL" DATA IN A COLUMN;
 DATA COLUMN;
 DO I = 1 TO _LIMIT;
   X = UNIFORM(0);
   FILE DD;
   PUT X 1-10 6.1;
 END;

*OUTPUT "REAL" DATA USING A FORMAT;
 DATA FORM;
 DO I = 1 TO _LIMIT;
   X = UNIFORM(0);
   FILE DD;
   PUT X 6.1;
 END;

FIGURE 10: LIST COLUMN AND FORMATTED OUTPUT

REFERENCES

1) CLASSICS IN SOFTWARE ENGINEERING
   Edited by E. Yourdon, (New York, YOURDON Press, 1979).

2) J. Sell, USING SAS FOR LARGE DATA SETS, SAS Technical Report A-104,

3) D. Henderson, SAS TUTORIAL: TABLE LOOKUP TECHNIQUES,
   SUGI 1982 Proceedings

4) D. Henderson, SAS TUTORIAL: SELECTING SUBSETS OF DATA,
   SUGI 1982 Proceedings

5) D. Henderson, SAS TUTORIAL: CONDITIONAL EXECUTION OF DATA AND PROC STEPS,
   SUGI 1982 Proceedings

6) J. Sharlin, IMPROVING THE DATA MANAGEMENT EXPERTISE OF EXPERIENCED SAS USERS, SUGI 1982 Proceedings

955