SAS Programming Efficiencies

Last revised: 04-21-99

The purpose of this document is to provide tips for improving the efficiency of your SAS programs. It suggests coding techniques, provides guidelines for their use, and compares examples of acceptable and improved ways to accomplish the same task. Most of the tips are limited to DATA step applications.

Efficiency

For the purpose of this discussion, efficiency will be defined simply as obtaining more results from fewer computer resources. When you submit a SAS program, the computer must:

- load the required software into memory
- compile the program
- find the data on which the program will execute
- perform the operations requested in the program
- report the result of the operations for your inspection

All of these tasks require time and space. Time and space for a computer program are composed of CPU time, I/O time, and memory.

CPU time the time the Central Processing Unit spends performing the operations you assign.

I/O time the time the computer spends on two tasks, input and output. Input refers to moving the data from storage areas such as disks or tapes into memory. Output refers to moving the results out of memory to storage or to a display device.

Memory the size of the work area that the CPU must devote to the operations in the program.

Another important resource is data storage - how much space on disk or tape your data use.

A gain in efficiency is seldom absolute because optimizing one resource usually results in increased consumption of another resource. For example, recreating a SAS data set in each run avoids the cost of storing the data set but increases the I/O and CPU time for the program. A few programming techniques do improve performance in all areas.

Execute only the Necessary Statements

The number and complexity of statements executed largely control the CPU time used. By default, SAS executes every statement in the DATA step for each observation in the input source. Within a statement, SAS executes every operation in a given expression each time the statement executes. Tips in this section fall into two main categories:

- tips that reduce the number of statements executed
- tips that reduce the number of operations performed in a particular statement.

Tip 1: When performing several calculations, be sure you execute them only for the necessary observations.

Acceptable:

```
libname read '~/dissert';
data temp;
      set read.adults;
      wkincome=rpincome/52;
      if age <50;
```

More efficient:

```
libname read '~/dissert';
data temp;
      set read.adults;
      if age <50;
      wkincome=rpincome/52;
```


By moving the subsetting IF statement before the recodes, the recodes are executed only for the observations that will appear in the output data set. This decreases the CPU time.

Tip 2: When only one condition can be true for a given observation, write a series of IF-THEN/ELSE statements.

Acceptable:

```
if division =1 then adjinc=rpincome/1.5;
if division =2 then adjinc=rpincome/1.4;
if division =3 then adjinc=rpincome;
if division =4 then adjinc=rpincome;
if division =5 then adjinc=rpincome/1.2;
if division =6 then adjinc=rpincome/1.2;
if division =7 then adjinc=rpincome/1.2;
if division =8 then adjinc=rpincome/1.3;
if division =9 then adjinc=rpincome;
if division gt 9 then adjinc=rpincome;
```

More Efficient:

```
if division =1 then adjinc=rpincome/1.5;
  else if division =2 then adjinc=rpincome/1.4;
     else if division =3 then adjinc=rpincome;
        else if division =4 then adjinc=rpincome;
           else if division =5 then adjinc=rpincome/1.2;
              else if division =6 then adjinc=rpincome/1.2;
                 else if division =7 then adjinc=rpincome/1.2;
                    else if division =8 then adjinc=rpincome/1.3;
```

```
                        else if division=9 then adjinc=rpincome;
                              else adjinc=rpincome;
```

In IF-THEN/ELSE statements, SAS stops checking statements when a condition is true for an observation. It then skips to the end of the series and then resumes processing. However, in a series of IF-THEN statements without the ELSE, SAS checks each condition for every observation. This tip is most useful when some categories contain many more observations than other categories and you can place the conditions in order of decreasing probability.

Tip 3: Perform resource-intensive calculations and comparisons only once.
Use IF-THEN/ELSE to test simple conditions.

Acceptable:

```
if division in(1,2,3) and year(date)< 1950 then
adjinc=rpincome/1.25;
else if division in(1,2,3) and year(date)< 1960 then
adjinc=rpincome/1.5;
else if division in(1,2,3) and year(date)< 1970 then
adjinc=rpincome/1.75;
```

More Efficient:

```
year=year(date);
if division in(1,2,3) then flag='y';
if flag='y' and year< 1950 then adjinc=rpincome/1.25;
   else if flag='y' and year< 1960 then  adjinc=rpincome/1.5;
      else if flag='y' and year< 1970 then  adjinc=rpincome/1.75;
```


Functions are expensive and require a lot of CPU time. By moving the calculations involving SAS functions, you have reduced the number of times the functions need to be evaluated per observation.

Tip 4: Assign a value to a constant only once.

Acceptable:

```
data scores;
   infile "scores.dat";
   input test1-test3;
   x=5;
```

More Efficient:

```
data scores;
   retain x 5;
   infile "scores.dat";
   input test1-test3;
```

SAS assigns values to variables in a retain statement only once. In contrast, SAS executes assignment statements during each iteration of the DATA step.

Tip 5: Put missing values last in expressions

Acceptable:

```
data test;
   set save.scores;
```

```
   /* t1 is often missing */
   score=t1 + t2/2 +t3/3 + t4/4;
```

More efficient:

```
data test;
   set save.scores;
   /* t1 is often missing */
   score=t2/2 +t3/3 + t4/4 + t1;
```

When a SAS expression contains several operations, a missing value propagates from the first operation in which it occurs through all subsequent operations in the expression. SAS records the column and line location of each use of the missing value and how many missing values occur at the location. The fewer the operations that involve missing values, the less record-keeping SAS must do and the less CPU time used.

Tip 6: Check for missing values before using a variable in multiple statements.

Acceptable:

```
data test;
   set save.sales;
   cost=whosale + oftmiss;
   tax=oftmiss*.05;
   profit=sales-oftmiss;
```

More efficient:

```
data test;
   set save.sales;
   if oftmiss ne . then do;
      cost=whosale + oftmiss;
      tax=oftmiss*.05;
      profit=sales-oftmiss;
   end;
```

Propagating missing values in expressions requires CPU time, as discussed in the previous tip. By checking for a missing value before performing the operations, you can reduce the number of missing values that are propagated.

Tip 7: Assign many values in one statement.

Acceptable:

```
if educ = 0 then neweduc="< 3 yrs old";
else if educ=1 then neweduc="no school";
else if educ=2 then neweduc="nursery school";
:
else if educ=16 then neweduc="Profess. degree";
else if educ=17 then neweduc="Doctorate degree";
```

More efficient:

```
proc format;
   value educf 0="< 3 yrs old"
1="no school"
2="nursery school"
3="kindergarten"
4="thru 4th grade"
```

```
5="thru 8th grade"
6="9th grade"
7="10th grade"
8="11th grade"
9="12th but nongrad"
10="H.S. Grad"
11="College,no degree"
12="Assoc.,occupat."
13="Assoc., academic."
14="Bachelor Degree"
15="Master Degree"
16="Profess. degree"
17="Doctorate degree";
run;

data new;
   set old;
   neweduc=put(educ,educf.);
```

When you must change many values of a variable to other values, using user-created formats with the PUT function enables you to make the changes in a single assignment statement, saving CPU resources.

Tip 8: Shorten expressions with functions.

Acceptable:

```
array c{10} cost1-cost10;
tot=0;
do I=1 to 10;
   if c{i} ne . then do;
      tot+c{i};
   end;
end;
```

More efficient:

```
tot=sum(of cost1-cost10);
```

Functions use precompiled expressions. Therefore, a DATA step containing a function needs to compile only the name of the function and its arguments. If you write an expression to do the same thing, SAS must compile all the operators and operands in the expression. This saves you CPU.

Tip 9: Use the IN operator rather than logical OR operators.

Acceptable:

```
if status=1 or status=5 or status=8 or status=9 then
newstat="single";
   else newstat="not single";
```

More efficient:

```
if status in (1,5,8,9) then newstat="single";
   else newstat="not single";
```

When SAS evaluates an expression containing the IN operator, it stops the evaluation as soon as a comparison makes the expression true. When SAS evaluates an expression containing multiple OR operators, it evaluates the entire expression even if one true

comparison has already made the comparison true.

Tip 10: Use a series of If-THEN clauses rather than compound expressions with AND.

Acceptable:

```
if status1=1 and status2=5 and status3=8 and status4=9 then
output;
```

More efficient:

```
if status1=1 then
   if status2=5 then
      if status3=8 then
         if status4=9 then output;
```

When a DATA step contains a series of conditions in IF-THEN clauses, the DATA step stops evaluating the series as soon as one clause is false. However, when the conditions are joined by AND, the DATA step evaluates all the conditions even if one is false.

Tip 11: Take advantage of SAS procedures.

Use procedures instead of code you develop yourself. This will save *you* time if nothing else. The following procedures produce output data sets that could be used instead of writing a data step to produce the data:

RANK	ranks the observations of numeric variables
STANDARD	standardizes variables to a specified mean and standard deviation
APPEND	appends data from one SAS data set to the end of another
TRANSPOSE	turns a SAS data set on its side
CORR	outputs a SAS data set containing a correlation matrix
FREQ	outputs a SAS data set of counts for variables or combinations of variables
MEANS SUMMARY UNIVARIATE	outputs summary statistics

Reading and Writing Data

Reading and writing data (I/O) is the largest single component of elapsed time in most programs, including SAS programs. Decreasing the number of I/O operations is the most effective way to improve the execution speed of your programs.

Tip 12: If several different subsets are needed, avoid rereading the data for each subset.

Acceptable:

```
data div1;
   set read.adults;
   if division=1;
run;
data div2;
   set read.adults;
   if division=2;
run;
data div3;
   set read.adults;
   if division=3;
run;
```

More efficient:

```
data div1 div2 div3;
   set read.adults;
   if division=1 then output div1;
      else if division=2 then output div2;
         else if division=3 then output div3;
```

Minimize the number of times you read large SAS data sets or external files by producing all the subsets you require for further processing in one DATA step. Test for conditions using IF/THEN statements and write observations to multiple data sets using OUTPUT statements. This tip saves both I/O and CPU time.

Tip 13: If a subset will be used only for analysis by another SAS step, use a WHERE statement rather than creating a new data set.

Acceptable:

```
data div1;
   set read.adults;
   if division=1;
run;
proc means;
   var salary;
run;
```

More efficient:

```
proc means data=read.adults;
   var salary;
   where division=1;
run;
```

When you want to run a procedure with only a subset of the data set, use a WHERE statement to select those observations rather than creating a subset data set and then running the procedure. This saves I/O, CPU time.

Tip 14: Read only the fields you need.

```
data adults;
   infile "census.dat";
   input @34  division $3.
         @112 rpincome 8.;
```

When you read records from an external file, you spend CPU time. By excluding unnecessary fields from the INPUT statement, you spend less CPU time, reduce I/O, and

save disk space.

Tip 15: Read selection fields first when inputting data.

Acceptable:

```
data adults;
   infile "census.dat";
   input @34  division $3. ...   @112 rpincome 8.;
   if division=1;
```

More efficient:

```
data adults;
   infile "census.dat";
   input @34 division $3. @;
   if division=1;
   input ... @112 rpincome 8.;
```

Determine if you can eliminate records based on the contents of one or two fields. Read those fields with an INPUT statement, using an @ line-hold specifier to hold the record, and test for a qualifying value. If the value meets your criteria, read the rest of the record with another INPUT statement. Otherwise, delete the record. Using this tip always saves resources with no tradeoffs or disadvantages.

Tip 16: Store data in SAS data sets.

Acceptable:

```
data adults;
   infile "census.dat";
   input @34  division $3.
         @112 rpincome 8.;
run;
proc means;
class division;
var rpincome;
run;
```

More Efficient:

```
libname read "~/dissert";
proc means data=read.income;
class division;
var rpincome;
run;
```

When you need to use SAS repeatedly to analyze or manipulate a particular group of data, create a permanent SAS data set instead of reading the raw data each time to create a temporary SAS data set. With your data already in a permanent SAS data set, you can avoid data steps altogether saving CPU time and extra I/O operations.

Tip 17: Store only the variables you need.

Acceptable:

```
data temp;
   set read.adults;
   if division =1;
```

More Efficient:

```
data temp(drop=division);
   set read.adults;
     if division =1;
```

There are variables you need only during DATA step execution. For example:

- index variables from DO loops
- variables holding values for testing conditions
- variables holding intermediate values in calculations

Eliminating these variables from the output data set saves disk space.

Tip 18: Process only the variables you need.

Acceptable:

```
data temp;
   set read.adults;
   year=year(date);
   if division in(1,2,3) then flag='y';
   if flag='y' and year< 1950 then  adjinc=rpincome/1.25;
   else if flag='y' and year< 1960 then
   adjinc=rpincome/1.5;
   else if flag='y' and year< 1970 then
   adjinc=rpincome/1.75;
run;
proc reg;
   model adjinc=year;
run;
```

More Efficient:

```
data temp(keep=adjinc year);
   set read.adults(keep=date division rpincome);
   year=year(date);
   if division in(1,2,3) then flag='y';
if flag='y' and year< 1950 then
adjinc=rpincome/1.25;
else if flag='y' and year< 1960 then
   adjinc=rpincome/1.5;
else if flag='y' and year< 1970 then
   adjinc=rpincome/1.75;
run;
proc reg;
   model adjinc=year;
run;
```

When you use data set options to select the variables you want, you eliminate all of the other variables from the program data vector as well as from the output data sets you produce. Using this tip can save you CPU time, I/O, and memory.

Tip 19: Use a null data set for writing external files.

Acceptable:

```
data temp;
   set read.adults;
   file "div1.dat";
   put @34  division $3.
       @112 rpincome 8.;
```

More Efficient:

```
data _null_;
   set read.adults;
   file "div1.dat";
   put @34  division $3.
       @112 rpincome 8.;
```

When creating external files using a DATA step, use the _NULL_ keyword in the DATA statement to run the step without creating a SAS data set. This saves CPU time, I/O, and disk space.

Tip 20: Shorten data using formats and informats.

Consider the following data:

```
00001 DIVORCED   1
00002 WIDOWED    0
00003 DIVORCED   2
00004 SINGLE     0
00005 MARRIED    5
00006 SINGLE     1
00007 MARRIED    0
...
```

Acceptable:

```
libname save "~/dissert";
data save.marital;
   infile "marital.dat";
   input id 1-5 status $ 7-14 no_chil 18;
```

More Efficient:

```
libname save "~/dissert";
data save.marital;
   drop status;
   infile "marital.dat";
   input id 1-5 status $ 7-14 no_chil 18;
   if status="DIVORCED" then marital="D";
   else if status="WIDOWED" then marital="W";
   else if status="SINGLE" then marital="S";
   else if status="MARRIED" then marital="M";
```

Even Better:

```
libname save "~/dissert";
libname library "~/formats";
proc format library=library;
   invalue $mar 'SINGLE'='S'
                'MARRIED'='M'
                'DIVORCED'='D'
```

```
                'WIDOWED'='W';
   value $mar   'S'='SINGLE'
                'M'='MARRIED'
                'D'='DIVORCED'
                'W'='WIDOWED';
run;
data save.marital;
   infile "marital.dat";
   input id 1-5 status $mar8. no_chil 18;
run;
proc print;
   format status $mar.;
run;
```

Define your own formats and informats when you can use a short code to represent long character data. Use informats to transform long character values in raw data into codes with shorter values in your SAS data set. Use formats that convert coded values to longer values when you need them. By using this tip, you trade an increase in CPU time for a decrease in storage requirements.

Tip 21: Use MODIFY instead of SET to modify existing variables or observations in a SAS data set.

Acceptable:

```
libname read  ~/dissert';
data read.salary;
     set read.salary;
     salary=salary*.1;
```

More Efficient:

```
libname read  ~/dissert';
data read.salary;
      modify read.salary;
      salary=salary*.1;
```

MODIFY does not make a copy of the data set like SET does. Only use MODIFY when disk space is an issue. This statement does not save I/O and uses more CPU. Note that you cannot create new variables with MODIFY.

Tip 22: Use the LENGTH statement to reduce storage space for variables in SAS data sets.

Acceptable:

```
data report;
   input local catalog;
   sales=local+catalog;
   cards;
100 400
300 800
:
;
```

More efficient:

```
data report;
   length local catalog sales 4;
```

```
   input local catalog;
   sales=local+catalog;
   cards;
100 400
300 800
:
;
```

SAS uses default lengths for variables in SAS data sets unless you specify a different length. For character variables and for numeric variables containing integers, you can save significant storage space by specifying the length. Caution: Do not shorten numeric variables containing fractions.

Using LENGTH statements slightly increases the CPU time.

Tip 23: Use character rather than numeric variables.

Acceptable:

```
data inventory;
   infile "inventory.dat";
   input item start finish;
   sales=start-finish;
```

More efficient:

```
data inventory;
   length item $ 4;
   infile "inventory.dat";
   input item $ start finish;
   sales=start-finish;
```

Use this tip only if you do not plan to perform arithmetic operations on variables. By default, numeric variables occupy 8 bytes of storage, whereas character variables can occupy as few as 3 bytes.

Tip 24: Use the COMPRESS= data set option when creating large SAS data sets.

Acceptable:

```
libname read  ~/dissert';
data read.salary;
   infile "salary.dat";
   :
```

More efficient:

```
libname read  ~/dissert';
data read.salary(compress=yes);
   infile "salary.dat";
   :
```

Compressed data sets require less storage space, and I/O operations are faster.

Tip 25: Use concatenation to reduce the number of variables you need to sort by.

Acceptable:

```
proc sort;
   by vara varb varc vard vare;
run;
```

More efficient:

```
data new;
   set old;
   sortvar=vara||varb||varc||vard||vare;
run;

proc sort;
   by sortvar;
run;
```

This second sort requires less storage space. Note that the concatenation operator (||) in SAS only works for character variables. If the variables you need to sort by are numeric, use the PUT function to convert them to character variables before the concatenation:

```
cage=put(age, 3.);
```

Tip 26: Use OUT= with the SORT procedure to write the sorted data to a different directory than the sort reads from.

Acceptable:

```
proc sort data=one;
   by age;
run;
```

More efficient:

```
libname save "~/sasstuff";
proc sort data=one out=save.one;
   by sortvar;
run;
```

The later sort requires less I/O operations because the sort reads from one disk (the work directory, /tmp) and writes to an other (~/sasstuff).

Tip 27: Use the DATASETS procedure to remove SAS data sets no longer needed.

Acceptable:

```
data one;
   infile "one.dat";
   input a b;
run;
data two;
   infile "two.dat";
   input a b;
run;
data all;
   set one two;
run;
proc sort;
   by a b;
run;
```

More efficient:

```
data one;
   infile "one.dat";
```

```
   input a b;
run;
data two;
   infile "two.dat";
   input a b;
run;
data all;
   set one two;
run;
proc datasets;
   delete one two;
run;
proc sort;
    by a b;
run;
```

This tip saves disk space.

Tip 28: When appropriate, distinguish between different types of missing values in the data.

Acceptable:

```
data survey;
   infile "survey.dat";
   input q1 q2 q3;
   if q3 in(-99, -999, -9999) then q3=.;
run;
```


Perhaps more appropriate:

```
data survey;
   infile "survey.dat";
   input q1 q2 q3;
   if q3=-99 then q3=.A;
      else if q3=-999 then q3=.B;
         else if q3=-9999 then q3=.C;
run;
```

This tip does not save resources but may result in more meaningful results with procedures like FREQ:

Y	Frequency	Percent	Cumulative Frequency	Cumulative Percent
A	5	6.7	1	6.7
B	1	6.7	2	13.3
C	1	6.7	3	20.0
1	35	6.7	4	26.7
2	40	6.7	5	33.3
4	95	6.7	6	40.0

Note that all these missing values still test the same in DATA step programs and that they are always treated the same in statistical procedures.